PERFORMANCE ANALYSIS AND OPTIMIZATION OF THE WINNOW SECRET KEY RECONCILIATION PROTOCOL

THESIS

Kevin C. Lustic, Civilian, USAF

AFIT/GCO/ENG11-08

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCO/ENG11-08

PERFORMANCE ANALYSIS AND OPTIMIZATION OF THE WINNOW SECRET KEY RECONCILIATION PROTOCOL

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Kevin C. Lustic, BS

Civilian, USAF

June 2011

AFIT/GCO/ENG11-08

PERFORMANCE ANALYSIS AND OPTIMIZATION OF THE WINNOW SECRET KEY RECONCILIATION PROTOCOL

Kevin C. Lustic, BS
Civilian, USAF

Approved:

/signed/	20 May 2011
Lt Col Jeffrey W. Humphries, PhD (Chairman)	Date
/signed/	20 May 2011
Michael R. Grimaila, PhD, CISM, CISSP (Member)	Date

Abstract

Currently, private communications in public and government sectors rely on methods of cryptographic key distribution that will likely be rendered obsolete the moment a full-scale quantum computer is realized, or efficient classical methods of factoring are discovered. There are alternative methods for distributing secret key material in a "post-quantum" era. One example of a system capable of securely distributing cryptographic key material, known as Quantum Key Distribution (QKD), is secure against quantum factorization techniques as its security rests on generally accepted laws of quantum physics. QKD protocols typically include a phase called "Error Reconciliation", a clear-text classical-channel discussion between legitimate parties of a QKD protocol by which errors introduced in the quantum channel are corrected and the legitimate parties ensure they share identical key material. This work improves one such reconciliation protocol, called *Winnow*, by examining block-size choices for *Winnow* and thus increasing QKD key rate. Block sizes are chosen to maximize the probability that each block contains exactly one error. Further analyses of *Winnow* are provided to characterize the effects of different error distributions on protocol operation and shed light on the time and communication complexities of the *Winnow* secret key reconciliation protocol.

Acknowledgements

Many thanks to MSgt Michael "Wooly" Woolingham and Capt Mark Duncan for their friendship and mentoring, and for helping me review and revise my research. I would also like to thank John for reviewing my source code and providing an optimization to my parity calculations, and for helping me with a side project while I was focusing on this research. Without the invaluable support of my advisor Lt Col Jeffrey Humphries and committee member Dr. Michael Grimaila, I would never have been provided with the opportunity to contribute to this intriguing and relevant project, and for this I am grateful. Finally I would like to thank my fiancée for supporting me in my academic pursuits at AFIT, and in all aspects of my life.

Kevin C. Lustic

Table of Contents

List of Figures

List of Tables

Conventions Used in This Paper

The terms *public channel* and *classical channel* are used synonymously to refer to an authenticated classical link between Alice and Bob used to carry out sifting, reconciliation, and privacy amplification. The term *raw key* refers to the random string of bits produced by Alice and sent to Bob over the quantum channel; the term *sifted key* refers to the string of key bits that are the outcome of the sifting process; the term *reconciled key* refers to the outcome of the reconciliation process; and the term *final key* refers to the outcome of privacy amplification.

The terms *estimated error rate* and *error estimate* are used synonymously, and in general differ from *error rate* or *actual error rate*.

PERFORMANCE ANALYSIS AND OPTIMIZATION OF THE WINNOW SECRET KEY RECONCILIATION PROTOCOL

I. Introduction

Cryptography is the art and science of protecting information by generating a new message, or ciphertext, such that interception of the ciphertext reveals minimal useful information to an adversary. Cryptography exists in many forms tailored to many applications including, but not limited to, protection of information in transit and in storage, proof of identity, and validation of information. For some applications, such as the secure storage of passwords on a web server or responses in a zero-knowledge proof authentication schemes, original information need not be recoverable by any party. For many others, namely those protocols designed to protect information communicated between multiple parties, the cryptographic scheme used to encipher data must facilitate the extraction of a meaningful plaintext.

There are countless cryptographic protocols available today that facilitate the confidential transmission of information. One type of protocol called *symmetric-key* encryption has many advantages such as speed of operation, relatively low power consumption, and in some cases an information-theoretic basis for security. One major shortfall of symmetric-key encryption is, as its name implies, that the sending and

receiving parties must both possess a copy of preshared key material in order to exchange meaningful messages. This issue may not seem critical, but one can imagine a scenario where sending and receiving parties wish to communicate securely but have not met prior to the information exchange, and hence share no private data for use as a key. For example, for transactions that take place over the Internet this is a major issue. The problem of establishing secret preshared keys for use in a symmetric cipher is commonly referred to as *key distribution*. To solve the problem of key distribution, symmetric-key protocols are often used in conjunction with *asymmetric-key* protocols, in which the sender and receiver do not share a common key but can still transmit a message securely. The security of asymmetric-key protocols in their current form is not provable. Instead these proofs typically rest on the limitations of modern computational resources. They are also generally inefficient for transmitting large messages due to time and/or energy constraints. However, such protocols are useful in their current form and context for sharing a relatively small key that two parties can use with a symmetric-key protocol for secure communications.

In the 1940's Claude Shannon, a researcher at Bell Labs, proved that an encryption protocol can only be unbreakable if the key used is truly random and is greater in length than the message to be encrypted, and no part of the key is ever reused (Shannon, 1949). The most well-known example of just such a protocol is the one-time-pad (OTP). Introduced by Gilbert Vernam of AT&T in the early twentieth century, the OTP was proven by Shannon to achieve perfect secrecy when implemented correctly (Shannon, 1949). Perfect secrecy refers to the fact that no information about a plaintext can be gathered from its ciphertext or, in terms of Shannon entropy $H(P|C) = H(P)$ where

P denotes the plaintext and *C* denotes the ciphertext. Due to its information theoretic security, OTP represents the ideal symmetric-key protocol from a security standpoint but, like other symmetric-key protocols, suffers the weakness of requiring key exchange in order to be rendered useful. Indeed, since the key required for use with OTP is the same length as the message to encrypt, the only known classical methods for establishing OTP keys between two geographically separated parties are by pre-sharing key material and by physical courier.

Therefore, modern cryptographic suites sacrifice some security in the interest of practicality. Symmetric-key schemes that require a reasonably small key are augmented with asymmetric, or public-key, protocols such as Rivest, Shamir, and Adelman's RSA algorithm for key distribution. The security of these public-key protocols is often based on the difficulty of solving problems that are currently considered mathematically difficult and computationally expensive. For example the security of RSA, which is used in online banking transactions, in the OpenSSH project, and a host of other Internet-based secure communication protocols, rests solely upon the perceived difficulty of factoring a large integer into two large prime factors. In fact, it is readily seen that an RSA ciphertext contains no information-theoretic security: because an RSA public key contains all of the information required to determine the private key, all the information needed to extract the plaintext from its ciphertext is available to an eavesdropper with extraordinary computing capabilities. While public-key protocols such as RSA are publicly secure within the context of today's computing capabilities, new techniques in factoring and distributed computing are advancing to the point of threatening these protocols. For recent progress in factoring semiprimes, see (Kleinjung et al., 2010).

While the best known factoring algorithms are currently still too slow to be used against RSA in practice, algorithms exist for full factorization in polynomial time on quantum computers (Shor, 1994). Experiments such as the factorization of 15 = 3 x 5 using Shor's algorithm on a quantum system show the realization of a device able to render our current key distribution techniques obsolete are not far-fetched (Politi, Matthews, & O'Brien, 2009). Such experiments underscore the need for secret key distribution techniques that are more robust than those based on computational complexity assumptions, and which are able to withstand an adversary that possesses a realized quantum computer.

Such cryptographic protocols have been studied for both the classical and quantum environment. Post-quantum cryptography, which is generally used to describe classical quantum-resilient protocols, includes hash algorithms used for authentication and public-key algorithms used for key distribution. For examples of progress in this field, refer to (Bernstein, Buchmann, & Dahmen, 2010). Also hypothesized for secure key distribution is quantum cryptography. This research focuses on a subset of the latter, namely *Quantum Key Distribution* (QKD). QKD exploits generally accepted laws of quantum physics rather than computationally hard problems to achieve key distribution, and in its pure theoretical form, quantum key distribution is secure against an infinitely powerful adversary. However, due to technological limitations, realizing ideal systems in the physical world is not possible and requires assumptions be softened. Thus practical implementations of quantum key distribution protocols require that legitimate parties of the key distribution protocols verify that they have obtained a truly secret and identical key. The steps of a quantum key-distribution protocol are shown in Figure 1, and are described in greater detail in subsequent chapters of this thesis.

The research presented in this thesis focuses on the *Reconciliation* phase of QKD, addressing critical parameters of the *Winnow* reconciliation protocol. The following questions highlight areas critical to the improvement and analysis of *Winnow*:

- How closely can Alice and Bob estimate the initial error rate without directly comparing a portion of the secret keys? How close must their error estimations be for *Winnow* to succeed? How can Alice and Bob affect this resolution?
- How can Alice and Bob choose an efficient block-size schedule given an estimated initial error rate? What are efficient block size choices?
- What effect do burst errors have on the operation of the *Winnow* protocol? Are these types of error distributions mitigated by shuffling the sifted key?
- How does *Winnow* perform on other metrics typically used to compare error reconciliation protocols?

The primary goal of this research is to determine and provide block size schedules for error rate estimates up to 0.20, which decrease the number of exposed bits of previously suggested methods. Block size choice is important for the optimality of *Winnow*. If the block size is too small, then Alice and Bob are likely to waste extra bits through parity comparison. However, if the block size is too large, then Alice and Bob may introduce errors into Bob's sifted key rather than correcting them. Therefore the goal is to find a block size that maximizes *Winnow*'s effectiveness. Finally, this research provides an evaluation of this author's implementation of the *Winnow* algorithm in terms of throughput; information leaked and thus discarded throughout protocol operation; threshold of failure for *Winnow*; and a treatment of the time and communication complexity of the *Winnow* protocol. The block size choice is defined theoretically and

determined empirically through Monte-Carlo simulation. All other topics are also investigated by Monte-Carlo simulation.

Chapter 2 presents background material on quantum key distribution and the *Winnow* reconciliation protocol, along with background material on two other popular Reconciliation protocols. Chapter 3 describes the experiments conducted to evaluate this author's implementation of the *Winnow* protocol, and provides a methodology for choosing block size schedules for *Winnow*. Chapter 4 gives the results of the experiments deigned and outlined in Chapter 3, along with a discussion of the results and their implications for the questions highlighted in this research. Finally, Chapter 5 gives a brief summary of the findings and how they impact quantum key distribution.

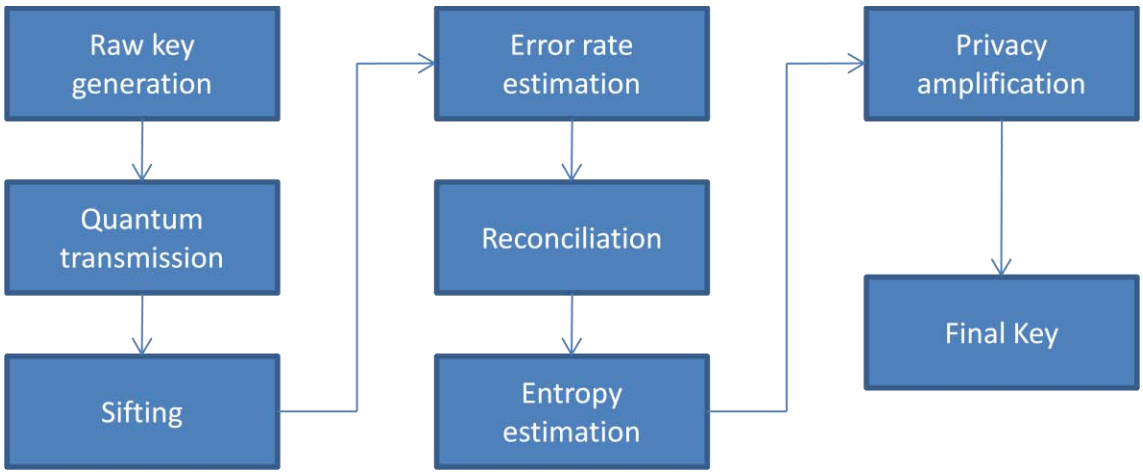


Figure 1. Progressive steps of a quantum key distribution protocol

II. Background

2.1. Quantum Key Distribution

In this chapter, a brief review of the principles of quantum physics is presented and literature relevant to the problem of error reconciliation, specifically the literature on *Winnow*, is presented.

While a classical Shannon bit exists as 0 *or* 1 a quantum bit, or qubit, can exist in a superposition of those two states. That is, a qubit is modeled in a 2-dimensional space, and thus any measurement of its state is probabilistic. Upon being measured, the qubit assumes a deterministic state that is correlated with the method of measurement; this is termed *collapsing* the state, an action which is synonymous with destroying the original state of the qubit. Much like a curve on a Euclidean plane can be projected onto the x- or y-axis, measuring the state of a qubit collapses said state onto one of its orthogonal 1-dimensional "axes". These axes, which together comprise the *basis* of the qubit, can be associated with the values 0 and 1 for encoding information in the qubit, and the qubit will attain one of these measurements with a probability that is associated with the original state of the qubit. Two bases are *conjugate bases* if they are mutually unbiased; that is, if a qubit is measured in its conjugate basis, the state of the qubit is collapsed onto one of the two orthogonal axes with equal probability and the information encoded in the initial state of the qubit is essentially lost.

In the late 1960's, a graduate student named Stephen Wiesner proposed the exploitation of quantum mechanics to create a novel means of communication (Wiesner, 1983). The concept was innovative, and because a part of the research required the

ability to store quantum particles for an arbitrary length of time, Wiesner's work did not gain much immediate attention and did not see publication until over a decade later. Wiesner's work proposed two ideas. One involves the creation of fraud-proof banking notes, which entails storing information in a quantum particle that could not be duplicated by a fraudster. The other, which Wiesner called *multiplexing*, proposed the transmission of two (or more) messages in such a way that reading one of the messages necessarily destroys the others. Wiesner's multiplexing involves encoding information in multiple conjugate bases of qubits, since observing the qubits in one of the bases implies the qubits are collapsed onto that basis and the data encoded in the other bases are lost. This notion of destroying information by mere observation motivated a new area of study in applied quantum physics, and not long after the publication of this work, quantum cryptography was born (Brassard, A bibliography of quantum cryptography, 1993).

The foundation of physical security for QKD is relatively straightforward. Quantum communication systems exploit the Heisenberg uncertainty principle, which implies that measuring a quantum system disturbs it and thus one cannot know everything about the state of the system prior to measuring it (Brassard, A bibliography of quantum cryptography, 1993). It is due to this principle that an eavesdropper unintentionally and unavoidably introduces flaws into a quantum channel which are detectable by the legitimate users. Therefore the security of a quantum channel rests, in theory, on fundamental laws of quantum physics rather than assumptions of the eavesdropper's computational power.

2.2. BB84 QKD Protocol

Though Wiesner did not apply his discovery of multiplexing to the field of cryptography, this application was quickly identified by Charles Bennett and Gilles Brassard. Bennett and Brassard described the advantage of Wiesner's multiplexed channels in cryptography; if an eavesdropper measures the state of a qubit improperly, she destroys the information encoded in that qubit. The implication is that this loss of information can be detected by the legitimate sender and receiver, who can consequently determine that an eavesdropper is present and listening in on the conversation. Bennett and Brassard subsequently proposed a system for sending messages assuredly free of passive and active eavesdropping, and noted the particular value of the system for the distribution of *cryptographic key material*. This protocol became known as the *BB84* quantum key distribution protocol, named after its authors and year of publication (Bennett & Brassard, 1984). The intention of BB84 is to utilize a quantum channel to distribute a random key the same length as the message to be encrypted, so that the key can be used as a OTP key. The combination of QKD and OTP, when implemented according to their respective assumptions, offers a cryptographic suite that is impervious to an infinitely powerful adversary. While achieving a OTP key is ideal, it is often unrealistic in practical contexts, and so a key long enough for use with another modern symmetric cipher that requires a smaller key can also be established with BB84.

In BB84, photons are used to realize qubits. Each photon is polarized using one of two mutually unbiased polarization bases. Bennett and Brassard chose the rectilinear basis, comprised of 0- and 90-degree polarizations, and the diagonal basis, comprised of

45- and 135-degree polarizations. A qubit polarized in one basis yields a random measurement in its conjugate basis.

The BB84 system is composed of a one-way quantum channel and a two-way classical channel. Owing to the physics of the channel, the quantum channel is subject to active eavesdropping but not passive eavesdropping. The classical channel is subject to passive eavesdropping but not active eavesdropping, if some initially shared secret material is used for authentication. The authors suggest the use of Wegman-Carter message authentication tags to satisfy this assumption on the classical channel (Wegman & Carter, 1981). The sender, who is responsible for generating and transmitting random key material and will hereafter be referred to as Alice, must have a way to generate single photons and polarize them in each of the polarization bases discussed previously. The receiver, hereafter referred to as Bob, is required to be able to detect and measure the state of single photons polarized by Alice. A high-level view of the communication channels between Alice and Bob is shown in Figure 2. In the figure, light lines represent a classical link and bold lines represent a quantum link. The arrows represent the possible flow of information, and legitimate users are Alice and Bob.

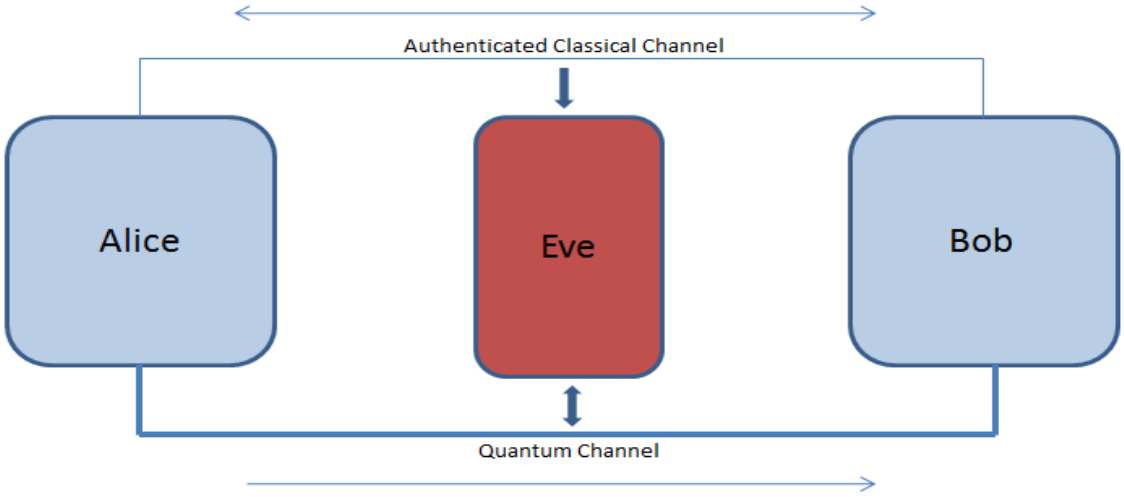


Figure 2. Actors and channels in BB84

When Alice and Bob wish to communicate a secret message, Alice generates and sends key material to Bob. She begins by generating random bits for key material, or *raw key*, and random bits for basis choice and the combination of basis and key bit choices determines the polarizations with which she will encode her qubits. For example, a key bit of zero would either be encoded in the rectilinear basis with a 0-degree polarization or in the diagonal basis with a 45-degree polarization, depending on the corresponding basis bit. A bit of 1 would be encoded in the rectilinear basis with a 90-degree polarization and in the diagonal basis with a 135-degree polarization. Once Alice has polarized a photon, she sends the photon over the quantum channel to Bob. Bob then chooses a random basis with which to measure the photon. He will choose the correct measurement basis one out of every two times on average. When all of Alice's key string has been transmitted, Alice and Bob communicate over the classical channel to determine which qubits were sent and measured in the same basis. On average, the qubits for which Alice and Bob choose mismatching bases are measured incorrectly with 50% likelihood, so all qubits for which the bases do not match are discarded. The process of publicly comparing bases to

find mismatches, as well as determining which bits Alice sent and Bob did not receive, is called *sifting*, and the remaining string of key bits shared by Alice and Bob is referred to as the *sifted key*. Ideally, in an environment in which no eavesdropper is present and the quantum channel is not susceptible to noise, once Alice and Bob have a sifted key then they share a random string that can be used as a secret key. Appendix A illustrates a BB84 QKD exchange without an eavesdropper present.

However, there would be no need for secure communication if there were no threat of an eavesdropper, Eve, listening on the quantum channel. Because the security of the channel lies in Alice and Bob's ability to detect Eve, it is essential to understand the impact Eve's activities have on the sifted key. The sifted key contains all bits for which Alice and Bob used matching bases. Assuming Eve intercepts and resends every qubit that Alice sends, she will randomly choose the incorrect measurement/sending basis with approximately 0.5 probability. Therefore, on average, the sifted key bits shared by Alice and Bob will include bits which Eve measures incorrectly with 0.5 probability and resends to Bob, who measures incorrectly with 0.5 probability.

To detect an eavesdropper, Alice and Bob select a random sample of bits from the sifted key and compare them over the classical channel to estimate the error rate. If the rate is unusually high, namely 0.5 x 0.5 = 0.25 or higher, they assume that there is an eavesdropper intercepting and resending their messages and start over.

There are other sources of error aside from an active eavesdropper. Error estimates are typically assumed to be correlated with the worst-case scenario of the presence of an eavesdropper, though it should be noted that many or all of the errors detected in a string of bits exchanged between Alice and Bob are typically due to

legitimate channel noise. In the literature covered by this research, the quantum channel is generally modeled as a *binary symmetric channel* (BSC) in which there is a certain probability *p*, representing the error rate, that the key bit being transmitted is flipped. In other words, if Alice sends the value 1 to Bob, then Bob will receive 0 with probability *p*. These errors are introduced by channel noise and other sources of interference. Due to the possibility of *legitimate noise* being introduced into Alice's message, an important note is that error rate estimates only translate roughly to an estimation of errors introduced by an active eavesdropper (Nakassis, Bienfang, & Williams, 2004).

Of course, as with any cryptographic protocol, the ideal assumptions of BB84 are limited in reality by implementation issues. For example, BB84 assumes the existence of a single photon source, quantum channels pure enough to carry the weak pulse generated by a single photon source, and detectors capable of detecting the weak pulse. Since technology in this area has not progressed to the point of satisfying these ideal assumptions, today's BB84 implementations require additional phases before a shared secret key can be extracted from the sifted key measured by Bob. These phases are *error reconciliation*, and *privacy amplification*. The following sections provide a brief treatment of the phases that take place after Alice and Bob complete the quantum transmission phase of QKD.

2.3. Sifting and Error Estimation

Sifting is the process of removing those bits from the key material for which Alice's sending basis and Bob's measuring basis do not match, and the result of sifting is the sifted key. Once the sifted key has been extracted by Alice and Bob, they proceed to

estimate the error rate of the channel. If the channel is considered a BSC with a chance *p* of Alice sending a bit and Bob obtaining the opposite measurement of the bit when measuring with the matching basis, then the error rate of the BSC is defined to be *p*. If *p* is abnormally high, Alice and Bob conclude the transmission was intercepted by an eavesdropper and start the key distribution over. More specifically, Alice and Bob decide on a threshold of acceptable error rate, say ε, such that if $p > \varepsilon$ they decide an eavesdropper is present. Assuming an eavesdropper measures and resends every photon in either the horizontal or rectilinear basis, ε should not exceed 0.25. If the error rate is 0.15 or higher then it is possible that an eavesdropper has intercepted and resent 60% or more of the transmissions; it is at this point that some believe the data should be discarded and the key exchange restarted (Nakassis, Bienfang, & Williams, 2004).

There has been a refinement of note that improves the sifting process in terms of throughput, by altering Alice and Bob's method for the selection of sending and measuring bases (Ardehali, Chau, & Lo, 1999). If the probability of choosing one basis for encoding a qubit is denoted σ and, hence, the probability of choosing the other basis is 1- σ, then the probability that Alice and Bob choose matching bases is given by the equation $P= \sigma^2+(1- \sigma)^2 = 2\sigma^2-2\sigma+1$. This probability effectively establishes an upper bound for the transmission rate of the channel, because Alice and Bob only keep the bits for which they choose the same basis. This probability function achieves a minimum when σ=0.5, the value used in BB84, which implies that the standard method for selecting a basis is theoretically the least optimal in terms of throughput. For a visual reference, a plot of P(σ) is shown in Figure 3.

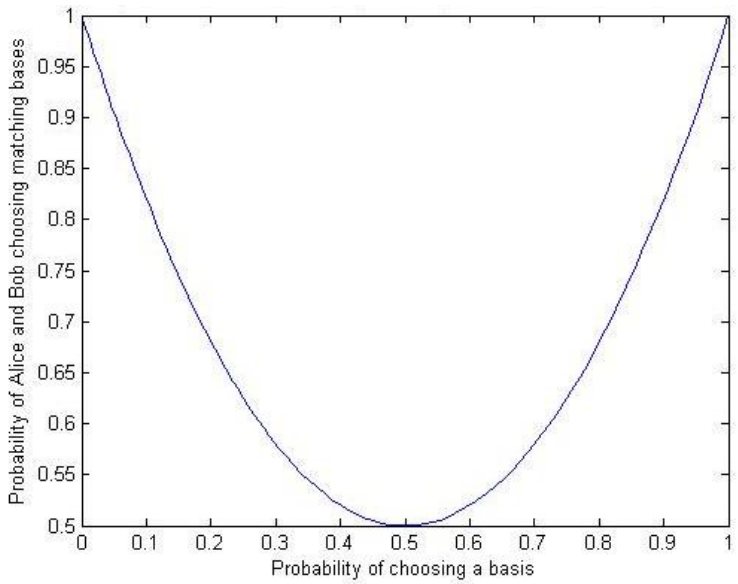


Figure 3. Probability of bases matching, as a function of basis choice probability

The authors propose using a different σ, with $0 < \sigma < 0.5$, to achieve a better upper bound on the transmission rate of the key exchange. For example the unbiased selection of $\sigma=0.5$ bounds the number of bits kept by 50%, while a biased selection of $\sigma=0.1$ bounds the number of bits kept by 82%. As seen in Figure 3, biasing the selection of one basis over another offers an increase in probability that Alice and Bob have selected matching bases, and thus there is an increase in the potential transmission rate of the channel using this simple technique.

However, an eavesdropper can also bias her measurement basis choice and thus, like Bob, be more likely to choose the same basis as Alice for any given qubit. This undermines the BB84 eavesdropper detection mechanism, which relies on the fact that Eve has no information about Bob's measurement basis choice before he announces it. To account for such a strategy, this biased basis selection technique also includes an alternate method of estimating error. In QKD protocols the error rate is typically estimated by selecting a random subset of the sifted key and publicly comparing, and

then discarding, the bits. Instead of choosing a random subset of all bits from the sifted key, Alice and Bob choose random subsets of the bits transmitted and received in *each basis*, and then estimate two basis-specific error rates. They then compare each error rate to ε individually. Theoretically, the error rates should be quite similar and should both lie below the established acceptable threshold. This error estimation strategy effectively eliminates the advantage Eve gains by choosing the more probable basis every time.

2.4. Reconciliation and Privacy Amplification

In any case, the transmission of qubits is largely affected by legitimate noise and non-idealities in the quantum channel regardless of whether an eavesdropper is present or not. In order to correct these errors without directly comparing their keys, Alice and Bob perform error reconciliation. There are two forms of reconciliation protocols, interactive reconciliation and one-way reconciliation. Interactive reconciliation protocols require that Alice and Bob interact via the public classical channel in order to detect and correct discrepancies, while one-way reconciliation protocols involve one party dictating how the second party should identify and correct errors. The public literature contains instances of both one-way and interactive reconciliation protocols. Three of the most popular protocols will be described in further detail below, namely *Binary*, *Cascade*, and *Winnow*. The first two are interactive protocols while *Winnow* is a one-way protocol.

The goal of reconciliation is to reduce the number of discrepancies between Alice and Bob's sifted keys in a way that guarantees Alice and Bob share an identical key while leaking as little information to Eve as possible. In *Binary* and *Cascade*, parity bits are exchanged between Alice and Bob to aid Bob in a binary search to detect discrepancies in

his sifted key. In *Winnow*, parity bits are exchanged to roughly locate errors, and syndromes of Alice's data are sent to Bob to pinpoint and correct the errors. Because these interactions are conducted over the authenticated public channel it is assumed that Eve can inspect but not jam the transmission, and so throughout the reconciliation phase Eve gains as many bits of information about the sifted key as the number of bits communicated between Alice and Bob. Specifically, if the number of bits sent between Alice and Bob is m, then Eve has gained m bits of information about the secret key (Van Assche, 2006). As reported in (Kollmitzer & Pivk, 2010), the minimum information Bob needs to correct his key to match Alice's is given by $n\mathrm{H}(p)$ where n is the length of the sifted key, p is the error rate, and H is the Shannon entropy function. Therefore a reconciliation protocol must reveal at least $n\mathrm{H}(p)$ bits and its efficiency in terms of information lost can be measured with respect to this bound.

Following reconciliation, privacy amplification reduces Eve's knowledge of the secret key gained during both quantum transmission and reconciliation to an arbitrarily small amount. More specifically, if the length of the reconciled key is L bits and it is estimated that a potential eavesdropped has gained m bits of information about the key, then privacy amplification should reduce the reconciled key to a length of L-m bits. If Alice and Bob instead reduce the length of their reconciled key to a length of L-m-s for a chosen parameter s, then Eve gains 2^{-s} bits of information about the key during reconciliation and her knowledge of the key can be made arbitrarily small (or large) by Alice and Bob's choice of s (Van Assche, 2006). Lastly, every output bit from the privacy amplification phase depends on a large number of the input bits. This bit

dependency implies that any information Eve has on Alice and Bob's reconciled key is masked by the information she does not have.

2.5. Three Popular Reconciliation Protocols

In 1991 five researchers, including Bennett and Brassard, wrote a paper together describing what they termed the "first experimental quantum key distribution channel ever designed and actually put together" (Bennett, Bessette, Brassard, Salvail, & Smolin, 1992). In their research they designed and implemented an interactive error correction protocol, hereafter referred to as *Binary*, which laid the foundation for further research into the improvement of the reconciliation stage of QKD protocols.

2.5.1. Binary.

In their experimental setup the authors, hereafter referred to as BBBSS, implemented the BB84 QKD protocol utilizing polarized photons in the rectilinear and circular bases. The implementation worked just as in the seminal BB84 research, substituting the diagonal basis for the circular basis, which is comprised of left and right spins of the photon and is conjugate to the rectilinear basis.

Once Alice transmits a stream of random qubits to Bob and both parties have obtained sifted keys, the sifted keys are presumed to contain discrepancies introduced by an eavesdropper and other sources of noise in the transmission channel. In order to assure that they share a common key, Alice and Bob perform a reconciliation protocol that detects and corrects discrepancies by binary search. This protocol, called *Binary*, works as follows.

Alice and Bob exchange a subset of the sifted key to estimate the error rate of the quantum channel. If the error rate is unnaturally high it may be due to the presence of an

eavesdropper, and Alice and Bob start over; this threshold is dependent upon the potential attack strategies of the eavesdropper. If the error rate is reasonable then Alice and Bob publicly agree upon a random permutation to apply to the bits in an attempt to rearrange the discrepant bits, hereafter referred to as *error bits*, into a relatively random distribution. Alice and Bob then divide the sifted key into blocks of N_1 bits, an integer value that is dependent upon the error rate. In general, N_1 should be chosen such that each block is expected to contain one or less errors on average. Next, Alice and Bob compare the parities of each block over the classical channel. If Alice and Bob find a block for which the parity disagrees, then that block contains an odd number of errors and they bisect the block into two sub-blocks of length $N_1 / 2$ and compare the parities of the first half to reveal which half contains the odd number of errors. By continuing in this way, Alice and Bob are able to locate the error bit after exchanging approximately $\log_2 N_1$ parity bits. In order to reduce the information about the key gained by an eavesdropper during reconciliation, Alice and Bob discard a bit from every block and subblock for which a parity was leaked, a process referred to as *privacy maintenance*. Once the errors of all blocks with disagreeing parities have been removed, Alice and Bob have finished the first pass of the protocol.

Note that any even number of (two or more) errors are not detected during the parity comparisons. So Alice and Bob permute the key again before performing a second pass of parity comparisons and binary searches. They continue in this manner for several passes, increasing the block size as the error rate of the sifted key decreases. Since Alice and Bob discard a bit for every parity they calculate and share, they effectively waste a bit for any block which doesn't contain any errors. So they switch to a more efficient

strategy called *confirm and bisect* when they are fairly confident that most errors have been removed. Rather than calculating the parities of every bit in the sifted key, Alice and Bob calculate and compare the parity of a random subset of the sifted key. If they find a subset for which their parity calculations do not match they perform *Binary* on that subset. After a sufficient number of random parity comparisons revealing no errors – BBBSS suggest 20 such comparisons – Alice and Bob are reasonably confident that their reconciled keys are identical. Figure 4 shows the detection and correction of an error. The number in each block/sub-block represents the corresponding parity of that block. Blocks/sub-blocks with the parity shown in italics represents a block for which a bit is exposed. Note that subdivided blocks do not contribute to the count of exposed bits, since the number of exposed bits in that block is the sum of those exposed in the subsequent step, and exposure estimation is deferred to that step. The detected error is shown with a strikethrough.

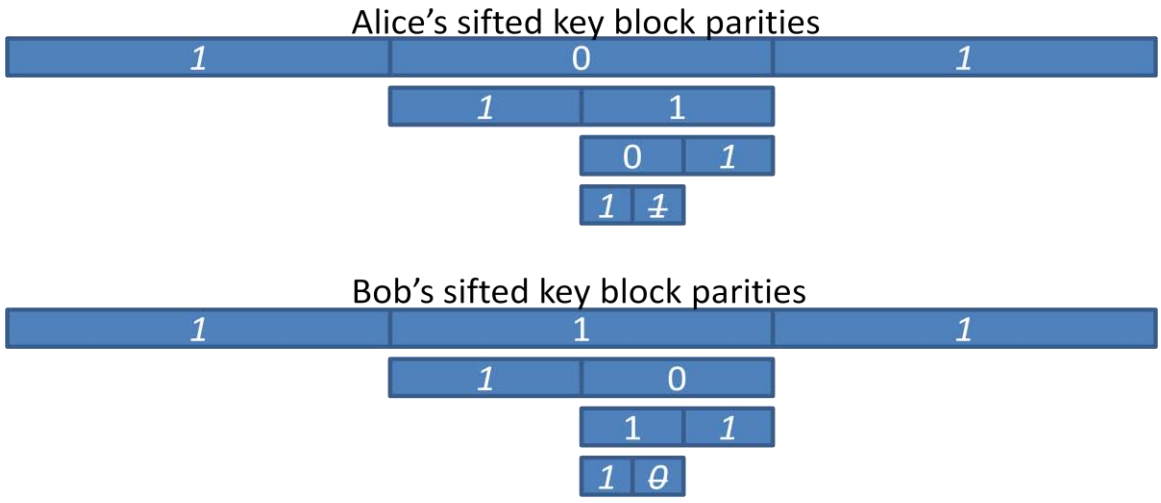


Figure 4. Correcting an error using *Binary*

It is readily seen that the number of bits Alice and Bob discard during *Binary* is given by $\sum_{i=1}^{t}\left(\frac{L}{N_i}+r_i\log_2 N_i\right)+s$, where t is the number of passes to performed, L is the length of the sifted key, N_i is the block size for the i^{th} pass, r_i is the number of errors corrected in the i^{th} pass, and s is the number of bits discarded during the final random sampling phase. Though the structure of the blocks is defined differently during the random sampling phase, s is similarly dependent upon the random block size, the number of errors detected and corrected, and the overall number of parities publicly compared.

2.5.2. Cascade.

Shortly after *Binary* was published two of its authors published a refinement to the protocol in which they claimed to reduce the number of bits exposed during reconciliation to near the theoretical minimum, with the tradeoff of requiring greater processing costs (Brassard & Salvail, 1994). Called *Cascade*, this protocol has the same basic structure as *Binary* with one major exception: rather than discarding bits as errors are detected and corrected, Alice and Bob retain all bits for use in future passes, which allows the algorithm to trace backwards.

Just as in *Binary*, *Cascade* begins with an estimation of the quantum channel's transmission errors by direct public comparison. Once they have estimated the error rate of their transmission Alice and Bob divide their sifted keys into blocks, the size of which depends on the error estimate. The choice of block size is important: if the blocks are too small, then many blocks will probably not contain errors and parity bits are needlessly exposed to Eve. However, if the blocks are too large then more passes must be performed to correct all errors, because one pass of *Cascade* only corrects at most one error per block, requiring excess parity leakage in another way. The authors of *Cascade*

empirically determined the ideal initial block size to be $N_1 \approx 0.73/p$ where p is the error estimate (Van Assche, 2006). Once parities are exchanged for each block, and the errors are corrected by performing *Binary,* Alice and Bob permute their bits and enter the second pass.

This is where *Cascade* differs from *Binary*. Alice and Bob permute the sifted key and increase the block size to $N_2 = 2N_1$ then perform *Binary* to locate and correct errors. However if Alice and Bob detect an error b_1, then because all parities in the previous pass indicated that there were no errors after the previous pass, they conclude b_1 masked an additional error b_2 in the previous pass. So they can locate b_2 using knowledge of the initial position of b_1. To locate b_2 Alice and Bob revisit the smallest sub-blocks for which b_1 was previously involved in a parity calculation, and proceed to bisect and locate the masked error from there. Unmasking that error may unmask other errors and so during any pass after the first, correcting a bit will cause the detection and correction of errors to *cascade* through previous passes. Correcting a bit in a later pass implies that two bits will be corrected in the sifted key, so the probability that any given block in the sifted key contains an error decreases exponentially as the protocol increments to a new pass (Brassard & Salvail, 1994).

Since its publication Cascade has seen numerous improvements in the public literature. One notable improvement greatly reduces the number of bits sent over the public channel by switching to a more efficient tactic after the second pass (Sugimoto & Yamazaki, 2000). The authors showed empirically that half of the errors were generally corrected after the first pass, then half of the remaining errors were corrected in the second pass and cascaded to correct the additional remaining quarter of the errors,

leaving only a few errors after just the second pass. Hence after two passes, *Cascade* could switch to *confirm and bisect* so that Alice and Bob compare a random parity rather than parities for every block in the sifted key. By this method, Sugimoto and Yamazaki were able to increase initial and subsequent block sizes and minimize the number of parities exchanged over the public channel. In doing so, the authors showed that their improvement leaked fewer bits to Eve and performed closer to the theoretical bound on throughput than the original *Cascade*.

Another improvement that can be made to *Cascade* is to modify the way bits are permuted between passes, in order to enhance the probability that two consecutive error bits do not mask each other in subsequent passes. Two methods covered in (Nguyen, 2002) are as follows:

1. Create a new block containing the first bit of each block. Then create a new block containing the second bit of each block. Continue in this way until each of the bits has been placed into a new block. Pair each block with a non-consecutive block and perform *Cascade*.
2. Let k_i be the block size for the next pass, and K_t the t^{th} block in the next pass. Then $K_t = \{b_u : u \equiv t \text{ modulo } k_i\}$, where u is the index of bit b in the previous pass.

Nguyen notes that this is rather inefficient for use in all passes, but will suffice to use between the first and second passes then revert to the original strategy of random permutations. Another method replaces the random permutation conducted between rounds with a more complex interleaving technique that attempts to mitigate "error sticking", or error clusters (Chen, 2000). Such methods are shown experimentally by

Nguyen and Chen to improve the performance of *Cascade* in terms of minimizing information leaked during reconciliation.

In addition to such tweaks as manipulating block size and block structure, the literature contains *Cascade* variants that globally revise the algorithm. One such variant goes so far as to completely do away with the error estimation phase that precedes the classical *Cascade* algorithm (Nakassis, Bienfang, & Williams, 2004). In this variant, the authors propose subdividing the sifted key and comparing respective parities until a crucial number of sub-blocks between Alice and Bob have discrepant parities. At this point, an error estimate is derived by means discussed in the research. Of note is the fact that this method of estimating the error rate can be much more efficient than, say, throwing away half of the bits by public discussion, as is performed prior to the normal operation of *Cascade*. The entire algorithm described by the authors is much different from *Cascade*, involving the jettisoning of entire blocks of bits for which the expected post-reconciliation yield is zero or negative.

2.5.3. Winnow.

Binary and *Cascade* are well-studied and their correction power and throughput have been improved in many ways, namely through the choice of block size given the error rate in the input string. However the literature seems to lack a rigorous proof for an upper bound on the information leaked to an eavesdropper by these protocols, and in fact the bound proposed by Brassard and Salvail can be exceeded experimentally (Yamazaki, Nair, & Yuen, 2006). In addition to the problem of bounding information leakage, *Binary* and *Cascade* suffer from an extremely high communication complexity. Owing to the interactivity of these two protocols, for which single parity bits must be sent

sequentially in order to correct a single error in each block, the number of messages sent to correct a single error in an N-bit block is $\log_2 N+1$. In a real-world implementation of *Cascade*, where network latency may be on the order of tens of milliseconds and a TCP/IP message adds 40 bytes of header information per segment (i.e. per *bit* sent for parity in *Cascade*), *Binary* and *Cascade* are highly inefficient when implemented over a TCP/IP network.

A reconciliation protocol providing solutions to both problems was proposed by W.T. Buttler et al. in 2003 (Buttler, Lamoreaux, Torgerson, Nickel, Donahue, & Peterson, 2003). Called *Winnow*, this protocol is based on Hamming codes. A short description of Hamming codes based on its treatment in (Hankerson, et al., 1991) is presented below, followed by the application of Hamming codes to reconciliation.

A Hamming code is a perfect code capable of correcting a single error and detecting two errors. For any Hamming code, there are two matrices called the *generator matrix* and the *parity-check matrix* which are responsible for encoding messages and detecting and corrected errors in the codewords, respectively. When a sender wishes to send a message, he computes the dot product of the message and the generator matrix and sends the resulting *codeword* to the receiver. The receiver calculates the *syndrome* of the received data by computing the dot product of the received data and parity-check matrix. If the data is a codeword then the syndrome is zero, as the term 'parity-check matrix' implies. However, if the syndrome is non-zero, then one or more errors have been introduced into the sender's codeword, and the syndrome identifies the coset leader associated with the codeword most likely associated with the received data. The codeword is the result of XORing the received data with the coset leader indicated by the

syndrome. Codewords are of length 2^m-1 for some *m* and have a distance of 3. This implies that Hamming codes can detect 2 errors, but may not detect three or more errors, and can correct $\frac{3-1}{2} = 1$ errors. In fact, if the Hamming algorithm is utilized to correct more than one error, an error may be introduced into, rather than eliminated from, the received data.

A few modifications are made to apply Hamming codes to the problem of reconciliation. First, the data from Alice is not encoded and sent directly to Bob. Doing so would make Eve's job much easier, as all she would need to do is decode the codeword to recover Alice's (secret) information, and thus this method would essentially compromise all of Alice's bits. Furthermore, Alice would be sending more bits of information than are contained in the initial data, so the entire private key would eventually be eliminated for privacy maintenance.

Instead, Alice calculates syndromes for her data and sends the syndromes. That is, Alice divides her key into blocks of length N=2^m-1 for some *m*, and constructs the parity-check matrix *H* where each element $H_{i,j}$ for $1 \leq i \leq m$ and $1 \leq j \leq 2^m$-1 is given by $H_{i,j} = \frac{j}{2^{i-1}} (mod\ 2)$. The syndrome *S* for a block B, then, is the product $H \bullet B = S$. Alice sends the syndrome to Bob, who computes the syndrome of his data in the corresponding block. If the syndromes match, that is S_A XOR $S_B = S = 0$, then Bob has either zero errors or three or more errors; for an appropriately chosen block size and a random error distribution Bob can be reasonably confident that there are no errors when S=0. If the syndromes do not match and Bob's string contains exactly one error, then *S* indicates the position in the string where the error can be found. Flipping this bit will correct the error.

Figure 5 presents an example of correcting a single error in Bob's data. Alice's data is represented by w_a and the corresponding syndrome is S_a, and Bob's data is represented with similar notation.

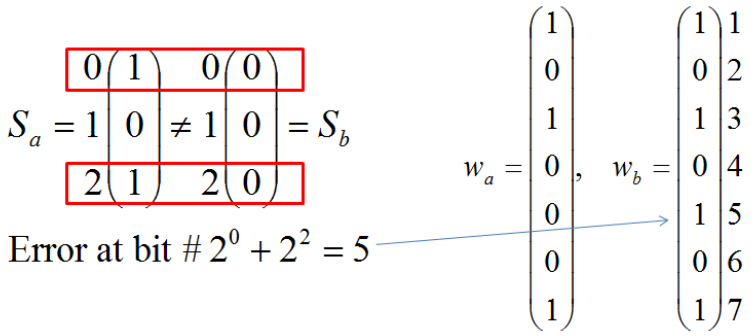


Figure 5. Correcting an error using Hamming syndromes

This is rather inefficient in terms of throughput as it implies that a single pass will expose approximately $m=\log_2 N$ bits per block, which will need to be eliminated for privacy maintenance. Recalling that *Winnow* only corrects an error in blocks containing *exactly one error*, another modification is made to reduce the number of bits exposed by the *Winnow* protocol by introducing a parity comparison prior to each pass. If the block size is chosen appropriately and the error bits are randomly distributed in Bob's key, there will likely be no more than one or two errors per block. Bob sends Alice the parity of each block, and Alice calculates the parities of her blocks. Alice then sends the syndrome of each block for which there is a parity error, that is, for which Bob's block contains an odd number of errors. In this way Alice will not expose *m* bits of information in the form of a syndrome if there are no errors in Bob's data, and Alice and Bob will not attempt to correct any even number of errors in general, since Hamming codes can never correct an even number of errors in any given block.

When one pass of *Winnow* is complete, the number of bits exposed for a block containing an odd number of errors is m+1 bits, and the number of bits exposed for a block containing an even number of errors is simply 1 bit. *Winnow* performs privacy maintenance throughout the reconciliation phase. Buttler et al. argue that this method, in contrast with the method of post-reconciliation maintenance used in *Cascade*, is better due to the fact that discarding bits during operation is likely to eliminate some errors unintentionally. The initial block size is modified to be $N=2^m$ for some m, since a bit is removed for parity comparisons before the syndromes are computed, making the block size the proper form for performing the Hamming algorithm. If a syndrome is transmitted, bits are also discarded after syndrome comparisons. The bits removed for syndrome comparisons are the bits whose positions in the block are of the form 2^k for $0 \leq k < m$. This selection is not arbitrary, rather these bits are the ones multiplied by the columns of the parity-check matrix of Hamming weight 1 and thus they are the bits that are maximally exposed in Alice's syndrome.

Because the parities and syndromes can be calculated for every block in parallel, and all single errors can be corrected in every block at once, a single pass of *Winnow* requires only two messages: a message from Bob to Alice containing the parity bits for each block and a message from Alice to Bob containing the syndromes for the blocks exhibiting parity errors. Thus the communication complexity of *Winnow* is vastly improved over *Cascade*. Furthermore, the amount of information exposed by Alice and Bob, along with the expected throughput of the algorithm, can be anticipated *a priori* in terms of the initial error rate. These models are presented in Buttler's research.

One further improvement is that, rather than publicly comparing and thus discarding a large number of bits for error rate estimation before *Winnow*, Alice and Bob begin the first pass of *Winnow* with a block size of 8 bits and determine the number of parity errors in Bob's data. The error rate *p*, then, can be estimated probabilistically based on that value. In the original research on *Winnow* Buttler et al. claim that knowing the initial error rate is sufficient for determining the ideal block size schedule given a set of data for successive passes, though a formula for finding the optimal block size schedule is omitted. The error rate estimation technique suggested by Buttler et al. is presented and analyzed in Chapter 3 of this research.

The most notable factors that influence the performance of *Winnow* are the block size and the distribution of errors. If a block contains an odd number of errors greater than one, *Winnow* may introduce a new error. This is in stark contrast to *Cascade*, which never introduces errors and always corrects an error in blocks containing an odd number of errors. Furthermore, if an even number of two or more errors exists and the errors are adjacent, they will never be corrected, which is a problem shared by *Cascade*. To address burst-error scenarios, a random permutation is applied to the sifted key prior to each pass in *Winnow* to attempt to shuffle error bits into a random distribution.

However, while *Winnow* provides a fast and efficient way to correct errors in two correlated random variables without direct comparison, its authors do not discuss a method for choosing ideal block sizes. Some improvements that have been applied to *Winnow* include the selection of block sizes in order to maximize throughput, that is, to maximize the amount of bits left after completing *Winnow*, or equivalently to maximize the amount of bits left and minimize the number of errors left. In one paper, a block size

of 8 bits is experimentally determined to have a higher "correction capability" than a block size of 16 bits. Hence, the ratio of $\frac{\text{number of errors eliminated}}{\text{number of bits discarded}}$ is greater after multiple passes with a block size of 8 than with a block size of 16, and so running *Winnow* with a fixed block size of 8 is better than running *Winnow* with a fixed block size of 16 for strings containing an initial number of errors in the range that are typical to today's QKD systems (Zhao, Fu, Wang, Lu, Liao, & Liu, 2007). Another study attempting to maximize correction ability through block size choice experimentally determined that the best choice for block size would be approximated by the ratio $Np \approx 0.8$, where *N* is the block size and *p* is the initial error rate (Yan, Peng, Lin, Jiang, Liu, & Guo, 2009). This result is quite similar to the choice of the initial block size for *Cascade*, for which Brassard and Salvail prescribe $kp \approx 0.73$ as reported in (Van Assche, 2006). For both these improvements a fixed block size is used for all passes, rather than a dynamic block size that accounts for the decreasing error rate like the doubling strategy in *Cascade*. In fact, aside from a few specific instances in the Buttler et al. research, the public literature does not include a treatment of the use of different (i.e. increasing) block sizes for successive passes in *Winnow*. Such a selection strategy would aid in improving the throughput of the *Winnow* reconciliation protocol.

III. Experimentation

3.1. Research Questions and Goals

While "quantum channel only" QKD is a promising future technology, the current technological readiness of publicly available QKD systems is such that public channel discussions remain essential. Central to the public channel component of QKD is the reconciliation phase. *Cascade* is currently the best known and most researched method for reconciliation, though the speed and communication efficiencies with which *Winnow* operates make *Winnow* a good competitor in certain operating conditions. And while researchers have analyzed and improved *Cascade*, similar analyses have not been applied to *Winnow*. Specifically, the public literature lacks a discussion of the choice of ideal block sizes given an estimated error rate, to maximize throughput while maintaining the speed and communication efficiency of *Winnow*. The following questions highlight areas critical to the improvement and analysis of *Winnow*:

- How closely can Alice and Bob estimate the initial error rate without directly comparing a portion of the secret keys? How close must their error estimations be for *Winnow* to succeed? How can Alice and Bob affect this resolution?
- How can Alice and Bob choose an efficient block-size schedule given an estimated initial error rate? What are efficient block size choices?
- What effect do burst errors have on the operation of the *Winnow* protocol? Are these types of error distributions mitigated by shuffling the sifted key prior to the first pass of Winnow?

– How does *Winnow* perform on other metrics typically used to compare error reconciliation protocols?

The primary goal of this research is to determine and provide block size schedules for error rate estimates up to 0.20, which decrease the number of exposed bits of previously suggested methods. A review of the public literature has not revealed a solution to this problem. Block size choice is important for the optimal efficiency of *Winnow*. If the block size is too small, then Alice and Bob are likely to waste extra bits through parity comparison. However, if the block size is too large, then Alice and Bob may introduce errors into Bob's sifted key rather than correcting them. The final research question is simply an evaluation of this author's implementation of the *Winnow* algorithm in terms of throughput; information leaked and thus discarded throughout protocol operation; threshold of failure for *Winnow*; and a treatment of the time and communication complexity of the *Winnow* protocol. Time and communication complexity are discussed in Appendix B.

Chapter 3 provides a brief description of the implementation of *Winnow* and the simulation environment, followed by a description of the experiments designed to answer these research questions.

3.2. Implementing *Winnow*

For this research, the *Winnow* protocol is implemented in C++ using two classes: one class is used to store and manipulate large buffers of bits efficiently both in memory and time, and the other class is used to perform the operations specific to *Winnow*, such as the calculation of syndromes and removal of bits. A third C++ file containing a main

function is used to drive the program, and generates a random sifted key. Errors are added to Bob's sifted key in accordance with the error rate specified by the user; this error rate is analogous to the error rate of the quantum channel, where the quantum channel is modeled as a binary symmetric channel. Therefore, the term "error" refers to a bit that is flipped during transmission. For randomly distributed errors, the errors are added such that each bit has an equal probability of being in error. For single burst errors, errors are introduced in a large block and the rest of the errors are distributed randomly. And for periodic burst errors, blocks of errors are introduced at a regular interval in accordance with the input error rate and size of the bursts.

Once the sifted keys are established, Bob computes his parities and 'sends' them to Alice. Alice then computes and compares her parities with Bob's, and for any blocks exhibiting a parity error Alice computes the syndrome of that block. Alice 'sends' a list of block numbers that exhibit a parity error along with the corresponding syndromes to Bob, who then computes his syndromes for the blocks of interest. If the syndromes match, Bob determines that one of two events occurred: either the error was discarded after comparing parities, or there are three or more errors in the block. In either case, it would not be beneficial to attempt to correct an error, and Bob does not attempt to fix any errors. Once Bob has fixed all feasible errors for the pass, Alice and Bob permute their bits in according to a publicly shared random permutation and proceed with the next pass. Minor tweaks are made to this process to facilitate experimentation as the particular experiment demands, but those changes are discussed in the following sections. Source code for this implementation of Winnow can be found in Appendix B and Appendix C.

The simulations are run on a Dell Latitude E6500 laptop with 4 GB of memory and an Intel Core2 Duo 2.66 GHz CPU. The operating system running on the laptop is Ubuntu Desktop 10.10 (Linux kernel 2.6.35-38). Development is done in Eclipse SDK and simulations for the following experiments are run from the command line. Status updates are printed to the screen, and statistical data are printed to tab-delimited files. Timing is done with a resolution of seconds.

3.3. Experiment 1: Error Rate Estimation

3.3.1. Determining error rate theoretically.

As proposed in the seminal research on *Winnow*, the error rate between Alice and Bob's sifted keys can be estimated probabilistically without publicly comparing and discarding a subset of the sifted key prior to reconciliation. Assuming each bit has an equal probability *p* of being in error, the probability of *n* bits being in error and *N*-*n* bits not being in error is given by $p^n(1\text{-}p)^{N\text{-}n}$. These errors can be arranged in multiple configurations within the block. There are "N choose n" different ways for *n* bits to be in error in an *N*-bit block, and so the probability of *n* errors occurring in an *N*-bit block in a sifted key with error rate *p* is estimated by Equation 1:

$$\binom{N}{n} p^n (1-p)^{N-n} \tag{1}$$

Hence, the probability of obtaining an odd number of errors in an *N*-bit block is represented by Equation 2:

$$\sum_{n_{odd}} \binom{N}{n} p^n (1-p)^{N-n} = \sum_{i=0}^{\left\lfloor \frac{N-1}{2} \right\rfloor} \binom{N}{2i+1} p^{2i+1} (1-p)^{N-(2i+1)} \tag{2}$$

If the first pass of *Winnow* is performed with a block size chosen independently of the initial error rate, then the ratio of blocks for which Alice and Bob have disagreeing parities or *parity errors* is known. That is, the probability that any block in a given set of data contains an odd number of errors is known. Setting this *parity error ratio* equal to Equation 2 gives Equation 3:

$$\frac{number\ of\ parity\ errors}{total\ number\ of\ blocks} = \sum_{i=0}^{\left\lfloor\frac{N-1}{2}\right\rfloor} \binom{N}{2i+1} p^{2i+1}(1-p)^{N-(2i+1)} \tag{3}$$

Intuitively, proceeding with the smallest block size of N=8 bits for the first pass will give the best resolution for the parity error ratio. For sufficiently small error rates a block size greater than 8 may be acceptable, but without *a priori* information about the error rate it is suitable to assume the error rate lies in the range typical of many of today's implementations of QKD systems, and an initial block size of 8 bits is the best choice in general (Buttler, Lamoreaux, Torgerson, Nickel, Donahue, & Peterson, 2003). Note that all the variables in Equation 3 are known but p, the error rate, and solving for p gives Equation 4:

$$p = \frac{1}{2}\left(1 - \left(\frac{B_{tot} - 2B_{pe}}{B_{tot}}\right)^{\frac{1}{N}}\right) \tag{4}$$

Where B_{tot} gives the total number of blocks, and B_{pe} gives the number of blocks exhibiting a parity error. In this way, Alice and Bob can estimate their error rate for any block size N.

3.3.2. Confirmation of estimation function.

After running *Winnow* in the simulation environment and gathering data on number of blocks exhibiting parity errors and total number of blocks, Figure 6 shows the parity error ratio as a function of actual initial error rate for a 1,000,000 bit input buffer and averaged over 50 trials.

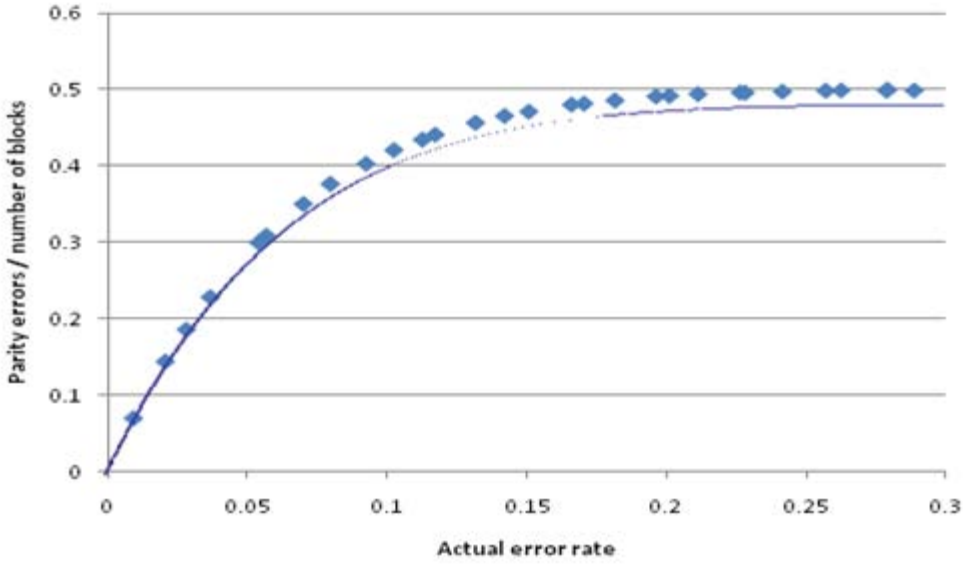


Figure 6. Relationship of parity error ratio and error rate

The curve represents the binomial distribution given on the right side of Equation 3 for N=8. Thus the experimental data confirms the accuracy of the theoretical estimation of error rate given in Equation 4.

3.3.3. Simulation parameters and factors.

Variables that affect the error rate estimation are the distribution of errors within the sifted key, block size, initial error rate, and length of the sifted key. The parameters that are manipulated for this experiment are error rate, which is controlled to be chosen randomly between 0% and 20%, and length of the sifted key, which is controlled to be chosen randomly between 10,000 and 10,000,000 bits. The block size is fixed at 8 bits,

as suggested by Buttler et al. The errors are assumed to be distributed randomly for this experiment.

3.3.4. Approach and methodology.

In order to test the effectiveness of the error estimation strategy given by Equation 4, a single pass of *Winnow* representing the first pass in a normal reconciliation effort is performed on two strings with a variable error rate and a fixed length. Once Alice and Bob have exchanged parity error information, Equation 4 is used to estimate the error rate based on the parity error ratio and total number of blocks. The recorded data includes the actual error rate, determined by an *Oracle* that has access to both Alice and Bob's sifted keys, and the estimated error rate. The metric used to compare the estimation with the actual error rate is by a factor of deviation, measured as:

$$deviation = \left| \frac{actual\ error\ rate - estimated\ error\ rate}{actual\ error\ rate} \right| \qquad (5)$$

This method of error estimation is statistically based, so another important question to answer is how large the input buffer must be in order to give a statistically significant error rate estimation. To attempt to answer this question, Experiment 1 is conducted again with the length of the input buffer made to vary. Once the buffer size is such that factor of deviation is consistently less than 0.10, the estimation is considered sufficiently close. For an explanation of this determination, refer to the results and discussion for Experiment 1 in Chapter 4.

3.3.5. Expected results.

Success is measured by plotting pairs of actual error rates and their corresponding estimated error rates and measuring the coefficient of determination of the resulting linear

regression. The regression is expected to have a slope of 1, and the coefficient of determination will be sufficiently close to 1 if the estimate is accurate. A conclusion is drawn about the necessary length of input size based on the deviation of the estimate from the actual error rate being sufficiently close. Suggestions are made for an input size for the remaining experiments, keeping in mind both accuracy of error rate estimation and simulation time.

3.3.6. Assumptions and limitations.

The inverse of Equation 3 is needed to find an equation for the estimation of p in Equation 4, however the inverse of Equation 3 is not a function. Therefore, Equation 4 exhibits asymptotic behavior that will limit the ability of this method to estimate error rates that are much greater than 20%. This argument can be visualized by referring to Figure 6; a critical parameter of the error estimation function presented in Equation 4, namely the parity error ratio, ceases to increase as the number of errors continues to increase beyond approximately 20%.

The symmetry exhibited by Figure 7 is not a major issue. Although an error rate of 90% will yield the same estimation using this function as an error rate of 10% Alice and Bob attribute such high error rates to a calibration error, and so if Alice and Bob recalibrate the error rate can always be assumed to be less than 50%.

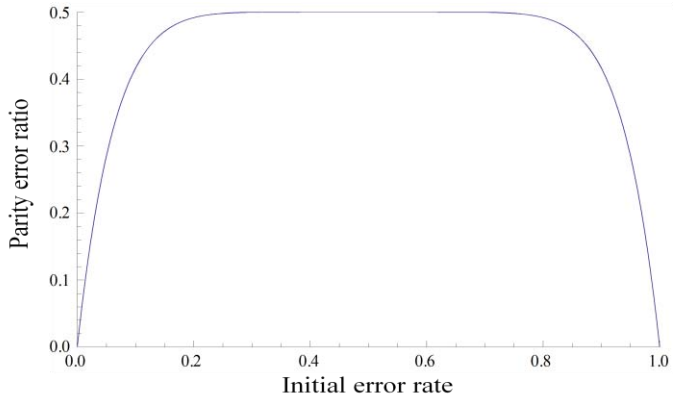


Figure 7. Parity error ratio as a function of error rate

Finally, because this method is dependent upon parity measurements its accuracy is highly dependent upon the normal distribution of errors. For example, an error rate of 0.10 may yield an estimate of 0% if the errors occur in pairs, or an estimate of 10% if the errors are evenly spaced. So this experiment assumes that errors are randomly distributed.

3.4. Experiment 2: Method for Dynamic Block Size Choice

3.4.1. Defining "ideal block size" theoretically.

Recalling that *Winnow* is based on Hamming codes, which are perfect single error correcting codes, the following observations can be made. If a block contains an odd number of errors greater than 1 then the efficiency is adversely affected; not only will $\log_2 N$ bits be discarded for the syndrome and 1 bit for the parity, but *Hamming* will likely introduce an error that will require correction in a later pass. Furthermore, if there is an even number of errors, one bit is discarded for parity and unless an error is discarded accidentally, no errors are corrected. Hence the power of *Winnow* can be fully utilized if a block contains exactly one error. Finally, if a block does not contain exactly one error, then *Winnow* will perform better on average if the block contains no errors than if the

block contains two or more errors. *Ideal block size* is used here to refer to the block size for which a block is most likely to contain exactly one error, given an estimation of the relative rate of errors in the sifted key. *Ideal error rate* is used to refer to the error rate for which a block of a given size contains exactly one error.

3.4.2. Selecting the ideal block size theoretically.

Following the reasoning by which Equation 1 is derived, the probability that an N-bit block contains exactly one error when the initial error rate of Bob's key is *p* is given by Equation 6.

$$p_1 = Np(1-p)^{N-1} \tag{6}$$

The goal is to maximize p_1 through choices of block size for any given *p*, and by the reasoning outlined above maximize the correction potential of *Winnow*. One example showing the block size choice which maximizes Equation 6 for an initial error rate of 0.10 can be seen in the plot of Equation 6, shown in Figure 8:

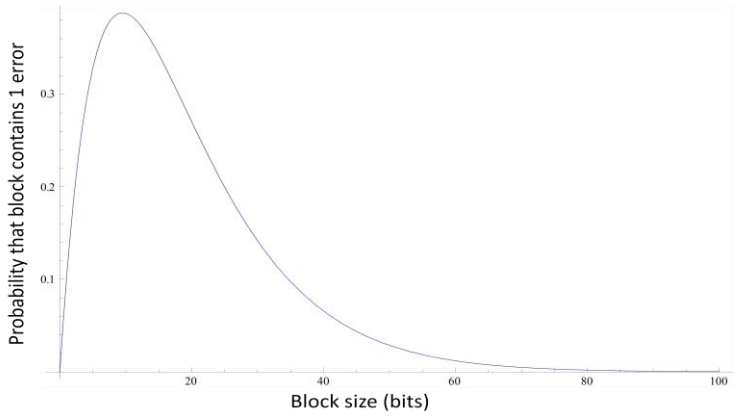


Figure 8. Probability of one error per block as a function of block size

While Figure 8 provides visual evidence of the ideal block size given an error rate of 0.10, it is advantageous to be able to compute the ideal block size given any error rate.

This can be done by setting the first derivative of Equation 6 with respect to N equal to zero and solving for *N*. The result is given by Equation 7:

$$N = -\frac{1}{\ln(1-p)} \tag{7}$$

For the example with an error rate of 0.10, the ideal block size is *N*=9.49 bits. In practice, due to the nature of Hamming codes, the block size must be a power of two, so the estimate given by Equation 7 can be rounded down to the nearest power of two to ensure one or less errors in a block.

Another way to view this result is to solve Equation 7 for the error rate *p* in terms of the block size. Equation 8 gives this result.

$$p_N = 1 - e^{\frac{-1}{N}} \tag{8}$$

The results are interpreted as, the ideal error rate for a block size of *N* bits is p_N. Table 1 provides the ideal error rates for block sizes 8 through 1024.

Table 1. Ideal error rates for each block size up to 1024 bits

Ideal Error Rates for Each Block Size	
Block Size (*N*)	**Ideal Error Rate (p_N)**
8	0.11750
16	0.06059
32	0.03077
64	0.01550
128	0.00778
256	0.00389
512	0.00125
1024	0.00098

Each ideal error rate can be interpreted as the threshold above which a block of size *N* is expected to contain more than one error on average, and below which a block of size *N* is expected to contain less than one error on average. So a strategy for selecting the ideal block size given an error rate *p* is to choose the block size *N* for which the inequality given in Equation 9 holds.

$$p_{2N} < p \leq p_N \tag{9}$$

If p is greater than p_8 then a block size of 8 bits should be used and if p is less than p_{1024} then a block size of 1024 bits should be used. The effective operation of *Winnow* requires an upper bound on the error rate. For a weak bound, Buttler and his team note that Hamming will introduce more errors than it will correct when $p \approx 0.30$. A stricter bound on this rate is not in the scope of this experiment.

3.4.3. Approach and methodology.

Determining the ideal block size schedule for a given initial error rate is done either dynamically, or using a lookup table. In this experiment, the block size schedule is first determined dynamically and then the resulting data comparing the error rate and appropriate block size schedule is analyzed to generate a lookup table. The process for dynamically determining the best next block size introduces a large number of floating point operations into *Winnow*'s code in practice, and thus while the throughput is expected to increase, the completion time is expected to show a marginal increase over a non-adaptive block size selection process.

In dynamically determining the best block size schedule, Alice and Bob begin with a block size of 8 bits for the first pass. After comparing parity bits, Alice and Bob can estimate the error rate between their sifted keys using Equation 4. Once they have

this information they can select the best next block size based on the error rate estimate, the values in Table 1, and the selection scheme in Equation 8. At the beginning of each pass Alice and Bob recalculated the error rate estimate and select the ideal block size for the next pass, keeping a count of the number of passes using each block size. The protocol terminates when Alice and Bob estimate an error rate of 0.

The lookup table, then, is generated from the results of the dynamic selection process. The reason for hard-coding block-size schedules is to remove most of the floating point operations of the dynamic method in hopes of maintaining the speed with which *Winnow* performs reconciliation, while also utilizing the findings of the dynamic method to achieve lower information leakage and thus higher throughput. The data from the dynamic method is sorted according to error rate, and divided into subsets representing a range of error rates. For example, all error rates p such that $0 < p \leq 0.01$ will use the block schedule for p=0.01. The ideal block size schedule lookup table is derived by applying the following rules to the data:

1. If the number of passes using a particular block size is frequently traded for the next larger block size, there is no need to do a pass with both block sizes. Select the maximum number of passes of the smaller block size and the minimum of the larger block size. This will slightly impact throughput but ensure that *Winnow* will correct all errors. An example derived from this research is given in Table 2 in Chapter 4.
2. For all other block sizes, use the maximum number of passes for which each block size was used in the dynamic method.

These rules ensure not only that error rates in the applicable range are covered by the ideal block size schedule, but also that the schedule works well for error rates at the low end of the next higher range.

3.4.4. Simulation parameters and factors.

For both the dynamic block size selection method and the lookup selection method the length of the sifted key, the initial error rate, the block size, and the distribution of errors are parameters that affect the ability of *Winnow* to locate and correct errors. The length of the sifted key is fixed at 1,000,000 bits, and the errors are distributed randomly throughout Bob's sifted key. The block size is determined by the algorithm for the dynamic selection method, and is a fixed input parameter for the lookup selection method. The factor for this experiment is the error rate, which varies randomly between 0% and 20% for each of 1,000 trials.

3.4.5. Expected results.

This dynamic selection method is expected to offer slightly higher throughputs over the lookup selection method at a small cost of speed. Throughput is defined as the percentage of sifted key remaining after performing reconciliation. The throughput is expected to be higher for sifted keys with a low error rate than for sifted keys with a high error rate, as more errors require more parity bits and syndromes to be exchanged, and thus exposed and discarded. This method is expected to perform better in terms of throughput than a non-adaptive method; the throughput is compared with such a method as well as the theoretical bound on throughput. The theoretical limit is proposed in (Kollmitzer & Pivk, 2010), where the minimum number of bits exposed by a reconciliation protocol is given by the Shannon entropy of the error rate. Note that for

Winnow, ‘exposed bits’ is synonymous with ‘discarded bits’ and so throughput represents the number of bits not exposed during *Winnow*. An example of a non-adaptive protocol is to fix the block size for all error rates to be 8 bits for all passes, as suggested in (Zhao, Fu, Wang, Lu, Liao, & Liu, 2007).

The number of passes for which each block size is used is not expected to be correlated with the initial error rate in general. The number of passes with a block size of 8 bits will likely increase as error rate increases. However, since the passes with 8-bit blocks will always drive the error rate below the ideal error rate for 8-bit blocks, a fixed value, before increasing the block size it is expected that the number of passes for which a block size of 16 or more bits is used will be very similar for all error rates.

3.4.6. Assumptions and limitations.

Note that error rates over 15% are considered unacceptable for BB84, as error rates of this magnitude can be attributed to the presence of an eavesdropper (Nakassis, Bienfang, & Williams, 2004). Therefore this method only seeks ideal block size schedules for error rates less than 0.20. In using these methods for selecting ideal block size schedules, Alice and Bob have the primary goal of ensuring all errors are detected and corrected and the secondary goal of maximizing the throughput of *Winnow*.

A limitation of this block size selection method is that its success is based on statistical assumptions of the distribution of errors within Bob’s sifted key, both for the estimation of the initial error rate as well as intermediate error rates, in the dynamic selection method. The methodology proposed for the dynamic selection method is relatively unaffected by the type of distribution of errors in the sifted keys, since the error rate is estimated continually during every pass and bits are randomly permuted between

each pass, but the lookup selection method is affected if the distribution of errors is non-random prior to the parity-based error estimation method described in Experiment 1. See Experiment 3 for a discussion of the effect of error distribution.

3.4.7. Advanced selection method.

This approach has one more limitation. The selection of the next block size is based upon Alice and Bob's estimation of the rate of errors in Bob's sifted key. This estimation occurs after parity bits are exchanged, but prior to exchanging syndromes. Therefore, once the error estimation is acquired it is too late to change the block size, and yet the actual rate of errors in Alice and Bob's strings will be altered before the selected block size is used. Therefore, while the block size chosen from the error rate estimation will be small enough and have a high enough correction power to eliminate errors, it may be smaller and hence less efficient than necessary. To test this hypothesis, a second round of experiments referred to as *Experiment 2 Variant* is performed using an identical methodology to that presented for Experiment 2, but for this variant an attempt will be made to estimate the number of errors that will be corrected by the *pass in progress,* and hence attempt to determine what the best block size *will be* after the pass is complete. To accomplish this more informed estimation, the following factors are considered when estimating error rate:

1. The error rate is first estimated per the method described in Experiment 1. This estimate represents the error rate prior to performing the current pass of *Winnow*.
2. The probability that a block contains one error is computed, and the probability that a block contains an odd number of errors other than 1 is

computed. The normalized values represent percentages of blocks exhibiting parity errors for each category. Consider these equations:

$$p_1 = Np(1-p)^{N-n}$$

$$p_{odd\ not\ 1} = \sum_{n_{odd}} \left(\binom{N}{n} p^n (1-p)^{N-n} \right) - p_1 \tag{10}$$

The fraction $p_c = \frac{p_1}{p_1 + p_{odd\ not\ 1}}$ gives the probability that an error is corrected in a block with a parity error, and 1-p_c gives the probability that an error is introduced under those conditions.

3. The probability that a block contains a nonzero even number of errors is computed. The fraction of blocks with an even parity will have an error unintentionally discarded for privacy maintenance after parity comparison with probability n/N, for a block size of N and an even number of errors n.

For simplicity, this is only done in passes with a block size of 8 bits. It is also assumed that the probability of discarding an error from a block with an even number of errors in Factor 3 is $2/N$, as there is only a 0.008 chance than an 8-bit block contains an even number of bits greater than 2. The results of Experiment 2 Variant are compared with the results of Experiment 2. Experiment 2 Variant is expected increase the throughput marginally over Experiment 2.

3.5. Experiment 3: Effects of Error Distribution on *Winnow* Operation

3.5.1. Potential types of error distributions.

This experiment quantifies the effect of the distribution of errors in the sifted key with regards to the effectiveness of *Winnow*, and the effectiveness of randomly permuting

the sifted key prior to running *Winnow* in mitigating any negative effects of non-random errors. Error distributions of particular interest are random distributions of errors, a single burst error among an otherwise random distribution of errors, and periodic burst errors.

Randomly distributed errors are introduced according to the philosophy that each bit has a probability equal to the input error rate of being in error. Therefore, randomly distributed errors may occur at regular intervals but may also exhibit small burst patterns.

Single burst errors are relatively large clusters of errors introduced into the sifted key. These bursts do not represent the total error rate, and the rest of the errors are introduced into the key in a random distribution around the burst. Periodic burst errors are errors that occur in the sifted key in clusters at regular intervals. Figure 9 shows examples of each of these types of error distribution.

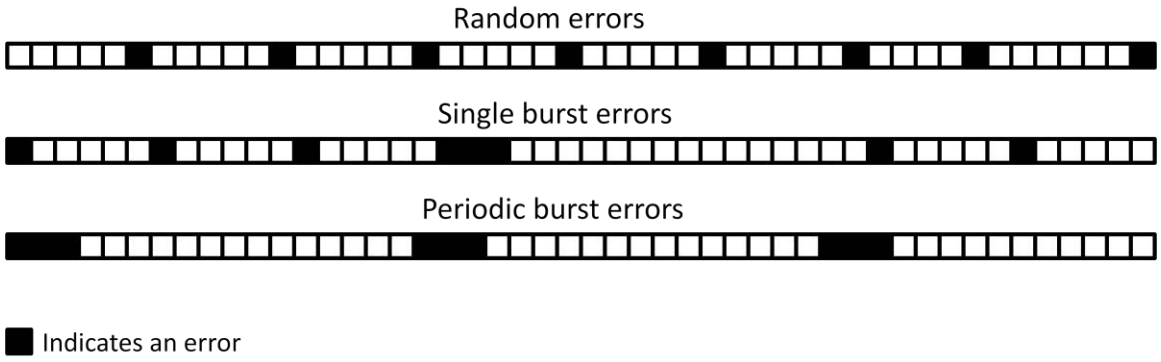


Figure 9. Types of error distribution

3.5.2. Simulation parameters and factors.

Parameters of Experiment 3 include length of the sifted key, type of error distribution, length of the error burst (if applicable), the selection of block size schedule, and input error rate. The length of the sifted key is fixed at 1,000,000 bits. For burst

error patterns the size of the burst is selected as a percentage that varies between 1% and 10%, of the total number of errors. Percentages of total errors are used rather than a fixed value so that the number of errors in a burst is relative to the total number of errors. For example, a 100-error burst intuitively has much less impact if the error rate is 15% than if the error rate is 1%, and this issue is overcome by using a burst size relative to the total number of errors. The initial error rate varies between 0% and 20%. The selection of the block size schedule is lookup-based and uses the lookup table from Experiment 2.

3.5.3. Approach and methodology.

Each burst size is simulated over 5,000 Monte Carlo trials, with and without initial random permutation of the input buffer. The output of the simulation consists of throughput and maximum errors left after performing *Winnow*. The introduction of randomly distributed errors involves flipping each bit in Bob's key with probability equal to the error rate. The introduction of single burst errors involves selecting a random starting position for the burst, introducing errors up to the burst length for that trial at that position, and then introducing the remaining errors evenly outside of the burst. The introduction of periodic burst errors involves dividing the total number of errors into bursts of the specified length and introducing these bursts at regular intervals into Bob's sifted key. The burst length is defined by a fraction of the total error rate of the sifted key; for example, if the error rate is 10% and the burst size specification rate is 0.05, then some 0.5% of the sifted key contains consecutive errors. For a visual example, in Figure 9 the burst size specification rate is approximately 0.3 and indeed approximately one third of the total number of errors occurs in each burst. This definition is chosen to

ensure that the actual burst size scales with error rate, where fixing a small burst size to work in both high and low error rates would likely not provide meaningful output.

3.5.4. Assumptions and limitations.

This experiment assumes that burst errors occur in perfect blocks. The periodic burst error experiment assumes that those perfect blocks occur in perfect intervals. Furthermore, the reader should note that this definition of *randomness* naturally generates burst errors anyway. For a series of N Bernoulli trials with two outcomes of equal probability, there is expectedly one string of $\log_2 N$ consecutive trials for which the outcome is the same. For example, out of 30 coin flips one can expect a string of 4-5 consecutive heads or tails. And in introducing errors randomly into an input of one million bits, which is a Bernoulli trial with a parameter p, one can similarly expect multiple consecutive errors. Therefore randomly distributed errors here are not expected to be uniformly distributed, and in general this error distribution may actually be interpreted as representing sparsely distributed and small single burst errors. This is not expected to affect the outcome of this experiment, and in general represents a better characterization of real-world QKD systems than would uniform errors.

3.5.5. Expected results.

For a random error distribution, *Winnow* is expected to operate normally, as it was designed to work under such conditions. For burst errors, *Winnow* is expected to fail to correct blocks containing burst errors on the first pass and is expected to perform sub-ideally. Blocks containing a large number of errors affect the operation of the *Winnow* protocol, such as Hamming introducing errors into blocks with odd number of errors greater than 1. Furthermore, burst errors are expected to affect the error estimation

phase, especially for periodic burst error patterns, which will affect the choice of block size and thus the correction ability of *Winnow*. This result will be reflected in the throughput and number of remaining errors. The failures induced by burst error distributions are expected to be rectified by applying a random permutation to the sifted key prior to performing reconciliation.

IV. Results and Discussion

The results of the Experiments described in Chapter III are provided below. Once the results are presented and conclusions drawn, some additional findings on the performance of this implementation of *Winnow* are presented.

4.1. Experiment 1: Error Rate Estimation

The plot shown in Figure 10 summarizes the data collected from the error estimation experiment for an input buffer length of 1,000,000 bits and error rates between 0% and 20%. The x-axis is the rate at which errors were introduced into Bob's sifted key, or actual error rate, and the y-axis is the estimated error rate. The data points represent 1,000 trials.

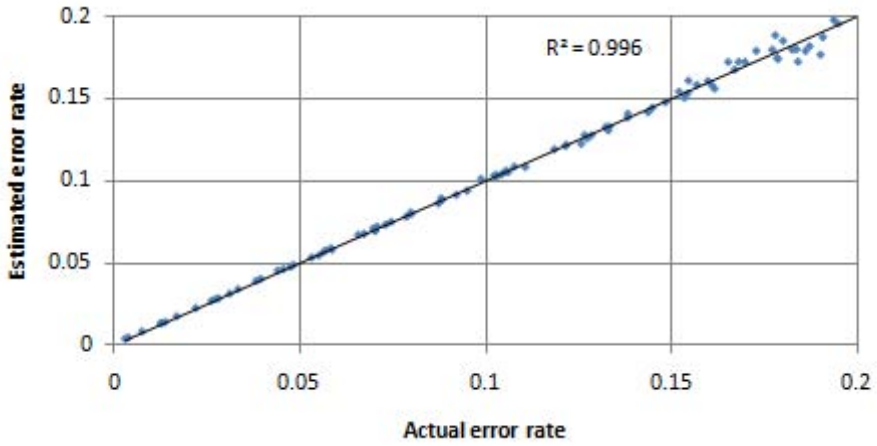


Figure 10. Estimated error rate compared to actual error rate

There is clearly a linear relationship. A linear regression fits the data visible well, and the coefficient of determination (R^2 value) indicates the line is a good fit for the range of values shown in Figure 10. However, this method of error estimation is not very

effective for actual error rates larger than 0.20. Figure 11 shows a plot of error estimation and the corresponding actual error rate for error rates ranging between 0% and 100%.

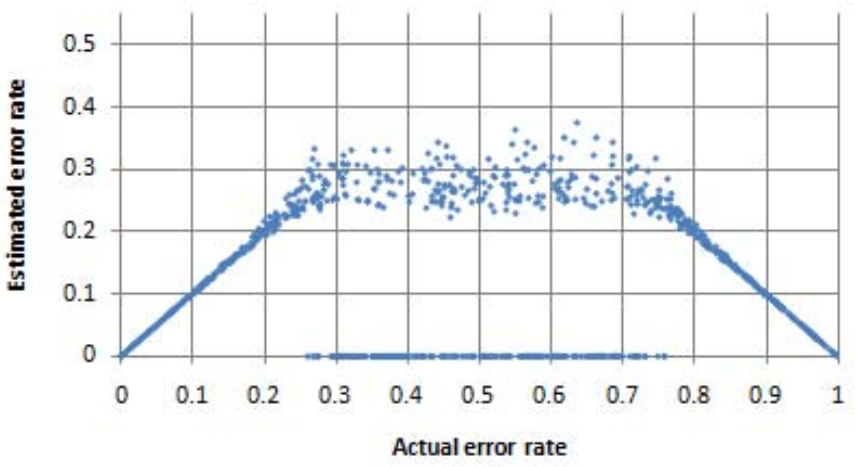


Figure 11. Estimated error rate compared to actual error rate

Not only is the correlation between actual and estimated error rates very weak for error rates between 25% and 75%, but in some cases the error rate cannot be estimated at all, which is represented by points lying along the x-axis. Furthermore, the estimated error rate is falsely reported for actual error rates greater than 75%. Error rates this high can be interpreted as calibration issues between Alice and Bob, and so in flipping every bit sent over the channel the rate can be kept below 50% errors and this issue of false reporting can be rectified. The problem of lack of correlation for error rates between 25% and 50% cannot be avoided. Therefore, Alice and Bob should start over if their error rates are estimated to be greater than 20%, or should use another method of estimation.

The size of the input also has a bearing on the resolution of the error rate estimate: too small an input buffer results in the estimation lacking statistical significance. However, for the purpose of simulating *Winnow* over large numbers of trials, the input must also be as small as possible. Figures 12 and 13 show the relationship between the

size of the input buffer and the closeness of the estimated error rate to the actual error rate.

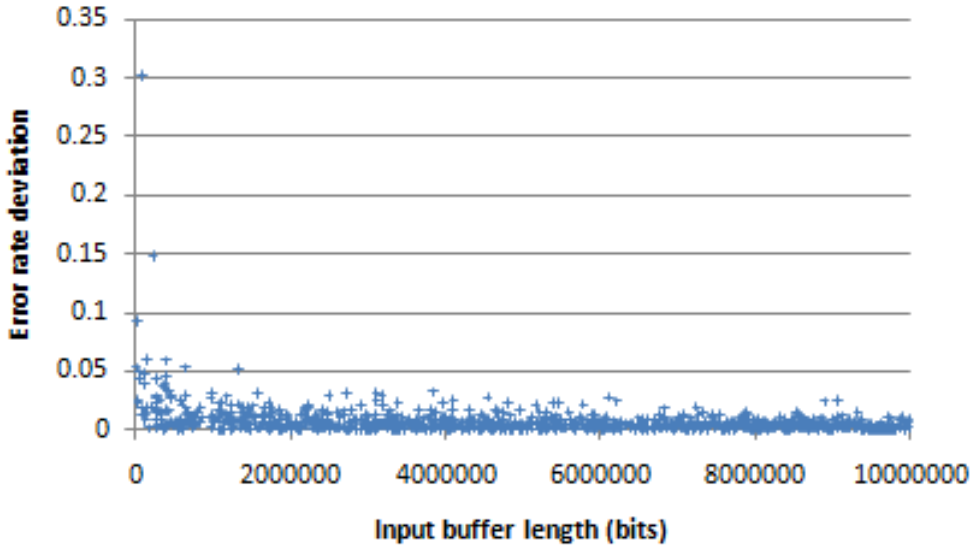


Figure 12. Deviation of error estimation as a function of input buffer length

With an input buffer length of 1,000,000 bits the deviation factor reaches a threshold of 0.10. This factor translates roughly to an increase of 1% in the difference between actual error rate and estimated error rate for every increase of 10% in actual error rate. Larger buffers provide better estimate resolution, but with diminishing returns and generally increasing runtimes.

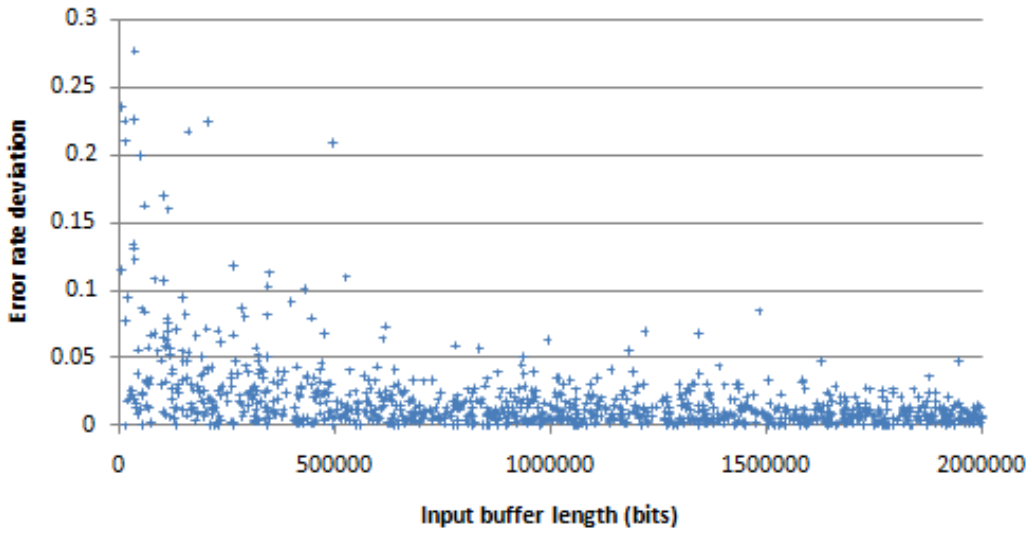


Figure 13. Deviation of error estimation as a function of input buffer length

In subsequent experiments, further data suggests this factor should be close to 0.05. For example, the results presented in Figure 17 indicate that Winnow tends to fail if the error rate estimation deviates by more than 1%. This is to be expected, as the block size choices determined in Experiment 2 are effective for a ±1% range in general. For an error rate of 10% the deviation factor would need to be 0.10 to keep the estimation difference below 1%. However, for an error rate of 20% the factor should be less than 0.05. Hence, using these findings, it is suggested that for statistical significance in the error rate estimation conducted prior to performing Winnow, the input buffer should contain 2,000,000 bits. Noting the sparseness of trials for which this factor exceeds 0.05 with an input buffer of 1,000,000 bits, as depicted in Figure 13, this length is considered *sufficiently large* for the remaining experiments in this research in order to keep simulation times low.

The data presented in this section answers the following questions:

How closely can Alice and Bob estimate the initial error rate without directly comparing a portion of the secret keys? How can Alice and Bob affect this resolution?

This is a function of the length of the sifted key that serves as input into the *Winnow* algorithm. The error estimation method is based on the assumption that errors are normally distributed, and the function used to estimate the error rate requires a large enough sample size to provide a statistically significant estimation. Depending on the input buffer size, one can approximate a "worst-case scenario" using Figure 12 and Figure 13. Alice and Bob can affect the resolution of the error estimation by buffering a larger or smaller amount of sifted key prior to performing Reconciliation with *Winnow*.

How close must their error estimations be for Winnow to succeed?

Using an initial block size of 8-bits, the data from this experiment suggests that one should have 1,000,000 bits of sifted key prior to error estimation in order to achieve an estimate that differs by a factor of 0.10 or less. Later experiments suggest that it would be desirable to have the estimate differ by no more than a factor of 0.05, and so it would be best for Alice and Bob to buffer 2,000,000 bits of sifted key prior to estimating error rate by the proposed method.

4.2. Experiment 2: Method for Dynamic Block Size Choice

The experiment provided 1,000 lines of tab-delimited data, including the input error rate, throughput, and chosen block size schedule. Results were sorted according to input error rate, and divided into 20 subsets each representing 1%error rate ranges between 0% and 20%. Each subset is assigned a block size schedule that works most

effectively for the error rates in that section. The patterns found in the data were remarkable.

As expected, an increase in error rate generally required an increase in correction power, thus an increased probability of choosing smaller block sizes. It was a pattern similar to the little-endian binary representation of decimal numbers, for which larger binary digits are required to represent larger decimal numbers. Consider the pattern shown in Table 2, which is a subset taken from the actual data. The full raw data and analyses can be found on the accompanying CD.

Some conclusions can be drawn from the data in Table 2, which was selected due to its representation of a transition in the need for a higher correcting power. This transition is evident in the variation between a block size of 64 bits and a block size of 128 bits for the middle range of rates shown, and also in the need for a second pass with a block size of 1024 bits when the algorithm chose a block size of 128 bits and perhaps should have chosen the stronger block size of 64 bits. The ideal schedule for error rates 0% to 1%, within which this data lies, was selected to be {1,0,0,1,0,0,1,2}, where each number represents the number of passes performed using block sizes of, respectively, {8,16,32,64,128,256,512,1024} bits. This selection is the product of the analysis conducted on the raw data in provided on the accompanying CD with respect to the rules outlined for Experiment 2 in Chapter 3. Note that this block size schedule will cover all the examples shown in Table 2, and covers all examples found for this range in the raw data.

Table 2. Experimental block size schedules as a function of error rate

Error Rate	Number of passes for each block size							
	8	16	32	64	128	256	512	1024
0.006502	1	0	0	0	1	0	1	1
0.006601	1	0	0	0	1	0	1	1
0.006679	1	0	0	1	0	0	1	0
0.006871	1	0	0	0	1	0	1	1
0.006927	1	0	0	1	0	0	1	1
0.006931	1	0	0	0	1	0	1	2
0.006955	1	0	0	1	0	0	1	1

The number of passes performed with a block size of 8 bits is generally proportional to the error rate, as can be seen in Figure 14.

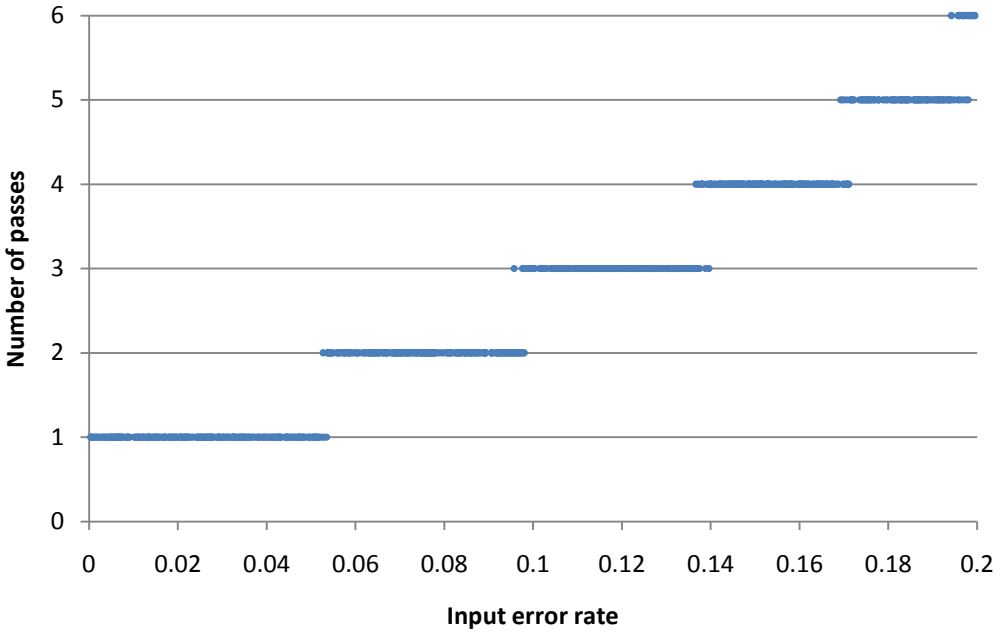


Figure 14. Number of passes with 8 bit blocks relative to error rate

However, the number of passes performed with a block size other than 8 bits is not generally proportional to the error rate. This result can be seen in Figure 15.

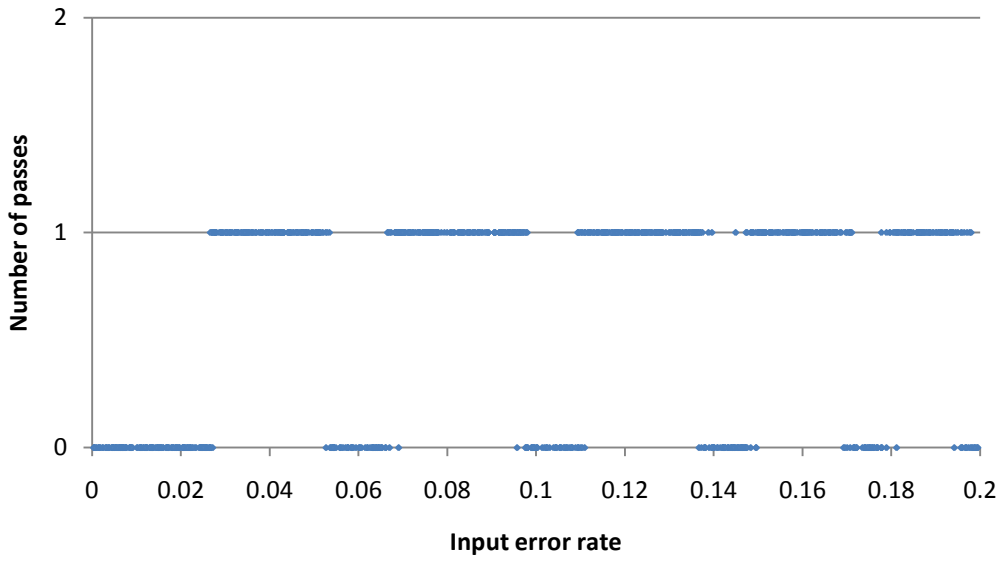


Figure 15. Number of passes with 16 bit blocks relative to error rate

It is interesting to note the correlation between the data presented in Figures 14 and 15. Particularly, transitions in the number of passes using a block size of 16 bits from one pass to no passes occurs when the number of passes using a block size of 8 bits increases. This speaks to the correction power of an additional pass with a block size of 8 bits reducing the need for a pass with a block size of 16 bits.

Using the experimental results from the dynamic selection method, the ideal block size schedules were derived and are summarized in Table 3. The error rate ranges in the first column of Table 3 specifies the range of error rates for which the corresponding block size schedule should be used. Of interest is the increasing tendency of the number of passes using 8-bit blocks. Also, most block sizes were used for only one pass or less before switching to a larger block size.

Finally, the throughputs of the two phases are shown among other data in Figure 16. The values are compared with the throughput of a non-adaptive block size selection

Table 3. Ideal block size schedules for error rates 0 - 20%

Error rate range (%)	Appropriate Block Size Schedule							
	8	16	32	64	128	256	512	1024
0-1	1	0	0	1	0	0	1	2
1-2	1	0	1	0	0	1	1	2
2-3	1	1	0	0	1	0	1	2
3-4	1	1	0	1	0	0	2	2
4-5	1	1	1	0	0	1	2	1
5-6	2	0	1	0	1	0	2	1
6-7	2	1	0	0	1	0	2	1
7-8	2	1	0	1	0	0	2	1
8-9	2	1	1	1	0	1	2	1
9-10	3	1	1	0	1	0	2	1
10-11	3	1	0	0	1	0	2	1
11-12	3	1	0	1	0	0	2	1
12-13	3	1	0	1	0	1	2	2
13-14	4	0	1	0	1	0	2	1
14-15	4	1	0	0	1	0	2	1
15-16	4	1	0	1	0	0	2	1
16-17	5	0	1	0	1	0	2	1
17-18	5	1	0	0	1	0	2	1
18-19	5	1	0	1	0	1	2	1
19-20	6	1	1	1	0	1	2	1

method as well as the upper bound on throughput for any reconciliation protocol. Throughput is defined to be the amount of remaining data after performing reconciliation, and thus the bound on throughput is derived from the fraction of bits discarded divided by the total number of input bits. The bound was determined in (Kollmitzer & Pivk, 2010). Interestingly, the large jumps in Figure 16 are correlated with the increase in number of passes with a block size of 8-bits; compare the position of the jumps with Figure 14. It is readily seen that the adaptive method for selecting block size schedules performs better in terms of throughput than the non-adaptive selection method. Furthermore, the dynamic selection method and the lookup selection method perform

very similarly. Finally, as the error rate approaches 20% the throughput of the adaptive method approaches the throughput of the non-adaptive method. This result can be explained by referencing Table 3, which demonstrates that the number of passes using an 8-bit block size generally tends to overshadow the number of passes using other block sizes as the error rate increases.

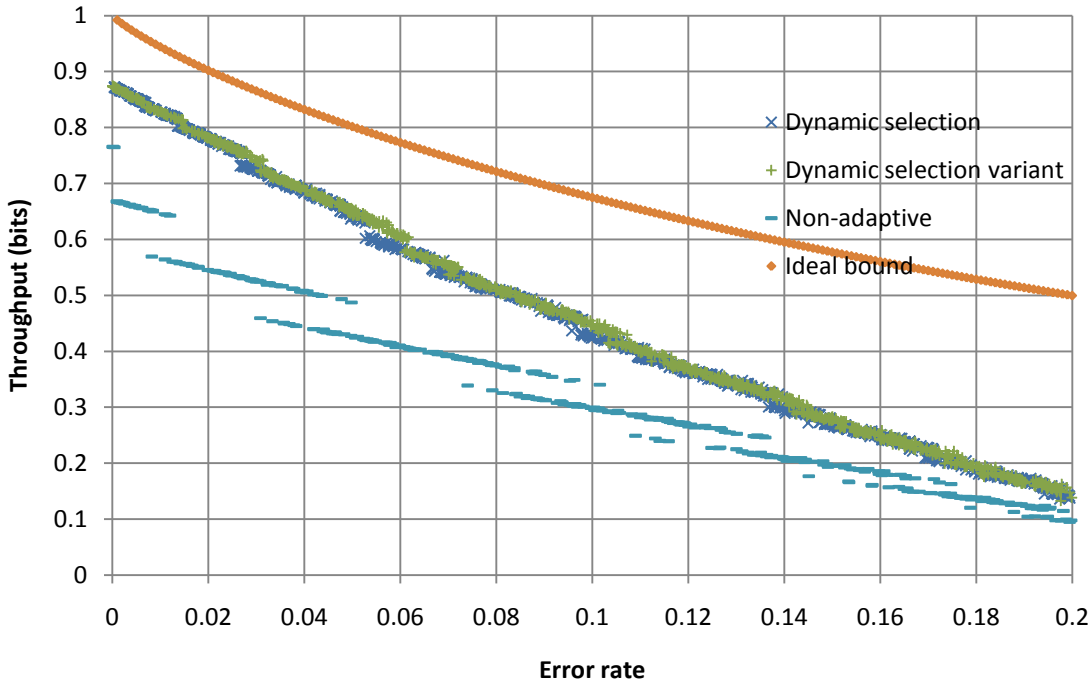


Figure 16. Throughput for *Winnow*

4.2.1. Experiment 2 Variant.

Using the same methodology described in the primary setup for Experiment 2, a different error rate estimation technique was employed to attempt to anticipate changes in the data. This method still relies on assumptions of a random error distribution, though it accounts for errors likely to be eliminated by privacy maintenance and Hamming

syndromes, as well as the potential for errors to be introduced when the number of errors per block is odd and three or larger. Between Experiment 2 and Experiment 2 Variant, the dynamic block size selection method of Experiment 2 Variant performed best in terms of throughput. The throughputs of both dynamic selection methods are plotted together in Figure 16. The other plots are described in the previous section.

The block size schedules determined in Experiment 2 Variant, shown in Table 4, vary slightly from the ones shown in Table 3. Of particular note is the reduced usage of 8-bit blocks, which increases the throughput of the algorithm at higher error rates. Furthermore, the method employed in Experiment 2 Variant led to a smoothing of the throughput curve. This can be attributed to the more effective use of larger block sizes by the algorithm employed in Experiment 2 Variant. For example, at the 9-10% error rate range, the Experiment 2 algorithm chose 8-bit blocks three times, whereas the Variant selected 8-bit blocks twice and rather than a third pass with 8-bit blocks added a pass with 64-bit blocks, thus wasting less bits for uneeded parity calculations. This impacted the throughput of the algorithm by approximately 2%, which is visible in the throughput plot shown in Figure 16.

The data presented in this section answers the following questions:

How can Alice and Bob choose an efficient block-size schedule given an estimated initial error rate? What are efficient block size choices?

One way, the proposed method, is to choose the block size such that the probability that any block contains exactly one error is maximal. Because *Winnow* is based on Hamming codes, which are single error correcting codes, it follows that *Winnow* performs best when a block contains one error. Furthermore, if a block does not contain

Table 4. Variant-generated ideal block size schedules for *Winnow*

Error rate range (%)	Appropriate Block Size Schedule							
	8	16	32	64	128	256	512	1024
0-1	1	0	0	1	0	0	1	2
1-2	1	0	1	0	0	1	2	1
2-3	1	1	0	0	1	0	2	1
3-4	1	1	0	0	1	0	2	1
4-5	1	1	0	1	0	1	2	1
5-6	1	1	1	0	1	0	2	2
6-7	2	0	1	0	1	0	2	1
7-8	2	1	0	0	1	0	2	1
8-9	2	1	0	1	0	1	2	1
9-10	2	1	1	1	1	0	2	1
10-11	3	0	1	0	1	0	2	1
11-12	3	1	0	0	1	0	2	1
12-13	3	1	0	1	0	1	2	1
13-14	3	1	1	0	1	0	2	1
14-15	4	0	1	0	1	0	2	1
15-16	4	1	0	1	0	0	2	1
16-17	4	1	1	0	1	0	2	1
17-18	5	0	1	0	1	0	2	1
18-19	5	1	0	1	0	1	2	1
19-20	5	1	1	0	1	0	2	2

exactly one error, it would be best if blocks contain 3 or more errors with only a very small likelihood, as errors will be introduced due to blocks containing an odd number of errors greater than 1. Maximizing the number of blocks containing exactly one error is accomplished by finding the block size that maximizes the binomial probability that a block contains exactly one error, given the most current estimated error rate. Finally, the error rate estimate can be made more realistic by estimating the number of errors corrected before the next pass, which necessarily affects the size of the block that should be used next.

Block size schedules generated by the proposed method are detailed in Table 3, and block size schedules generated by the second method, which includes estimating the number of errors corrected by a given pass, are given in Table 4.

4.3. Experiment 3: Effects of Error Distribution on *Winnow*

4.3.1. No permutation applied prior to the first pass.

For the first portion of Experiment 3, the sifted keys were not permuted prior to the first pass of *Winnow*. As expected, the algorithm still performed well for randomly distributed errors, and corrected all errors with the typical throughput and in the typical amount of time. However, introducing burst errors into Bob's sifted key led to utter failure of the protocol. This was mainly caused by the failure of the error estimation method, as can be seen by correlating the difference between the estimated and actual error rates with the number of errors left after *Winnow*. Refer to Figure 17 for a visualization of this result for single burst errors.

The error rate was typically underestimated; this is to be expected, because burst errors will tend to mask each other and thus not be considered in the parity-based error estimation. Furthermore, as the error rate estimation deviated much further than 1% from the actual error rates the number of errors not corrected by *Winnow* increased. The data in Figure 15 represents the data produced by single burst errors with no permutation applied prior to performing *Winnow*. The outcome for periodic burst errors was far worse, and could not be presented graphically in a useful manner. In general the error rate estimation was approximately 0%, as all errors masked each other for estimation, and thus too weak a block schedule was chosen a majority of the time; almost all errors remained uncorrected for non-permuted periodic burst errors.

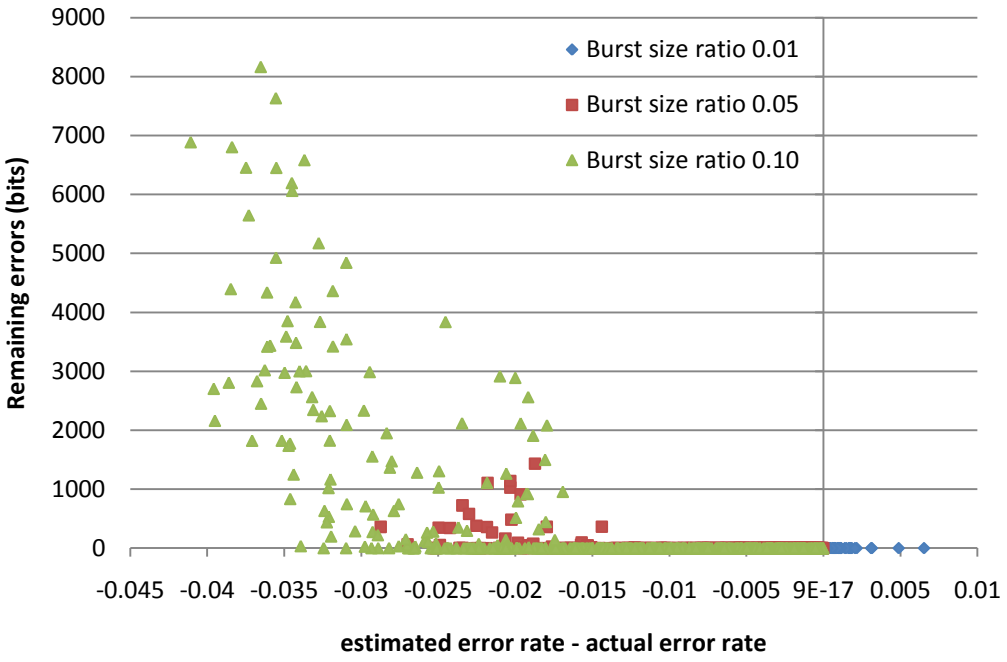


Figure 17. Remaining errors as a function of error rate estimation difference

4.3.2. Applying a permutation prior to the first pass.

Applying a random permutation before the first pass rectified these shortcomings; in all but one trial out of 5,000 total trials for single burst errors, and all but two trials out of 5,000 total trials for periodic burst errors, *Winnow* was able to correct all errors. The permutation did not affect the performance of *Winnow* for randomly distributed errors. Figure 18 shows the throughput for *Winnow* for inputs with randomly distributed errors, inputs with single burst errors, and inputs with periodic burst errors and with a random permutation applied prior to the first pass. The burst size for the burst error patterns in Figure 18 is the largest used for experimentation, namely 10% of all errors.

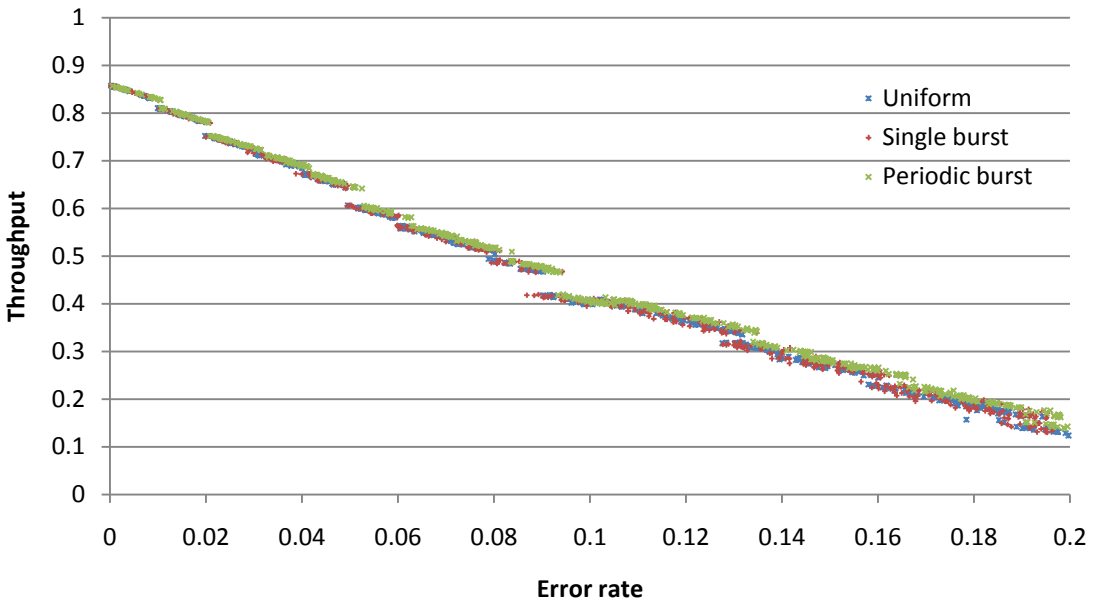


Figure 18. Throughput for *Winnow* with different error distributions

To summarize, *Winnow* performs extremely poorly for burst error distributions unless a random permutation is applied prior to the first pass. Permuting the sifted keys randomly prior to reconciliation fixes the failure of *Winnow* in nearly all trials, and does not adversely affect the performance of *Winnow* for randomly distributed errors.

The data presented in this section answers the following questions:

What effect do burst errors have on the operation of the Winnow protocol?

Single burst errors and periodic burst errors do affect the operation of the *Winnow* protocol when no permutation is applied to the bits prior to the first pass. In most cases, these types of error distributions affect the error rate estimation, causing too low an estimate by reducing the number of blocks exhibiting parity errors in Equation 4. If the error rate is estimated too low, then an inappropriately low-powered block-size schedule will be selected and Winnow will fail to correct all errors. For periodic burst error distributions, *Winnow* almost always estimated an error rate of 0%. For burst error

distributions, the estimation deviated by an amount that was generally related to the size of the burst. The number of errors not corrected at the end of *Winnow* as a function of error rate deviation can be seen in Figure 17 for single burst errors with three different burst sizes.

Are these types of error distributions mitigated by shuffling the sifted key prior to the first pass of Winnow?

It appears that they are, and that errors that are normally distributed prior to such a permutation strategy are not rearranged into a burst or otherwise harmful distribution. This result is to be expected, as *Winnow* performs best when errors are randomly distributed.

V. Conclusions

Current methods for secure communication include symmetric protocols and asymmetric protocols. Asymmetric protocols, or public key protocols, are typically used to distribute key material for use in symmetric key protocols. The asymmetric protocols of today generally rely on the computational unfeasibility of performing difficult mathematical operations, so-called trapdoor functions. Many of these mathematically hard problems, to include factoring semiprimes and computing discrete logarithms, have been solved for quantum computers and as the prospect of quantum computing becomes increasingly promising, the need for a more secure means by which to distribute key material becomes increasingly apparent. While "post-quantum" cryptographic protocols address this issue using classical components, several quantum key distribution (QKD) protocols also exist with a security foundation based on generally accepted laws of quantum physics.

These QKD protocols are still quite nascent, and as such the ideal assumptions of ideal components have yet to be realized. One consequence of this shortcoming is the addition of an error correction phase inserted into a QKD protocol in order to ensure that the communicating parties share a common final key. One such protocol which enjoys a relatively small communication complexity is called *Winnow*, and this research addresses the issue of finding ideal block sizes for use in the *Winnow* reconciliation protocol.

In one competing protocol, *Cascade*, it is suggested that the communicating parties publicly compare a subset of their bits in order to estimate the error rate of the

sifted keys prior to error reconciliation. Buttler et al. argue that this isn't necessary and propose an alternative means of error estimation, under the assumption that errors are normally distributed. This error rate estimation technique performs extremely well, and the communicating parties can increase the accuracy of the method by increasing the number of bits in the sifted key prior to error estimation. In fact, if the communicating parties accrue two million bits of sifted key before estimating the error rate, then the estimation will on average be within a factor of 0.5% of the actual error rate, which is sufficiently close to reduce the probability of failure of *Winnow* to a negligible amount. Furthermore, by randomly permuting the bits prior to the first pass of *Winnow*, the communicating parties can ensure the maximal effectiveness of the error estimation technique, and of the *Winnow* protocol.

While the normal distribution of errors is central to the effective operation of *Winnow*, the selection of block sizes used by the protocol is also quite important. For blocks containing an even number of errors, a parity bit is unnecessarily sent and thus discarded. For blocks containing an odd number of errors greater than 1, *Winnow* can actually introduce errors. Therefore an ideal block contains exactly one error. This fact can be exploited to select ideal block sizes, once an error rate is known. That is, once the communicating parties have a decent estimation of the number of errors in their sifted keys, they can use Equation 7 to determine the best block size for a given error rate, or similarly Equation 8 to determine the best error rate for each block size that is suitable for use in *Winnow*. Figure 19 demonstrates the concept of too large or too small a block size. This research provides a methodology for an adaptive block size selection scheme, and the results are summarized in Table 3 and Table 4.

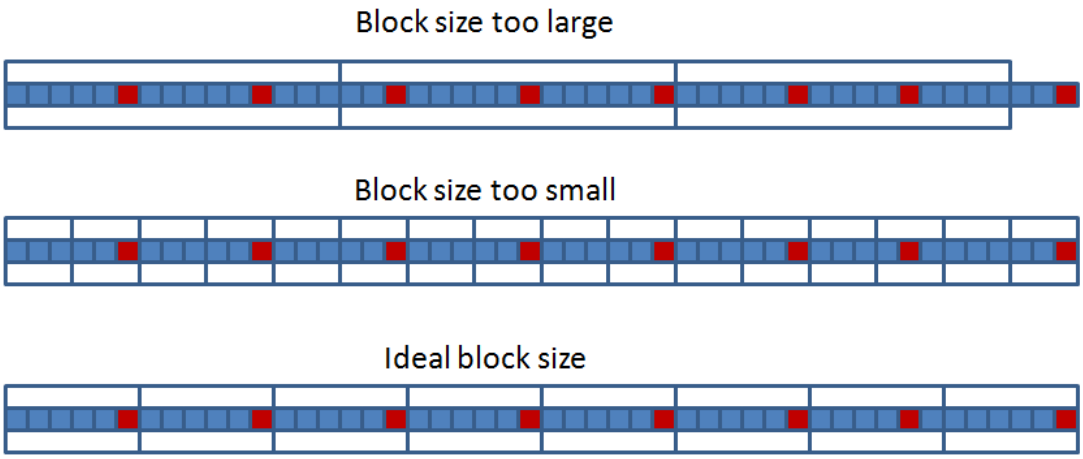


Figure 19. Visual representation of ideal block size

Winnow performs well with normally distributed errors, but due to the fact that burst errors can be introduced into the channel by environmental interference and other sources of both periodic and asymmetric noise, a study of the effect of burst errors on the operation of *Winnow* is treated in this research. Of particular interest are single burst errors, in which a controlled percentage of errors occur in a large burst and the rest of the errors occur randomly around the burst, and periodic burst errors, in which all errors occur in burst distributed in regular intervals in Bob's sifted key. *Winnow* has the ability to recover from single burst errors in this scenario unless the error rate estimation deviates by more than a factor of one percent from the actual error rate. This can be seen visually in Figure 20. Periodic burst error patterns lead to the complete failure of *Winnow*. Applying a random permutation to the sifted keys prior to performing *Winnow* fixes this shortcoming, and in the experiments performed for this work applying a random permutation prior to the first pas of *Winnow* led to throughputs for periodic and single burst error patterns which were similar to the throughput for normally distributed errors.

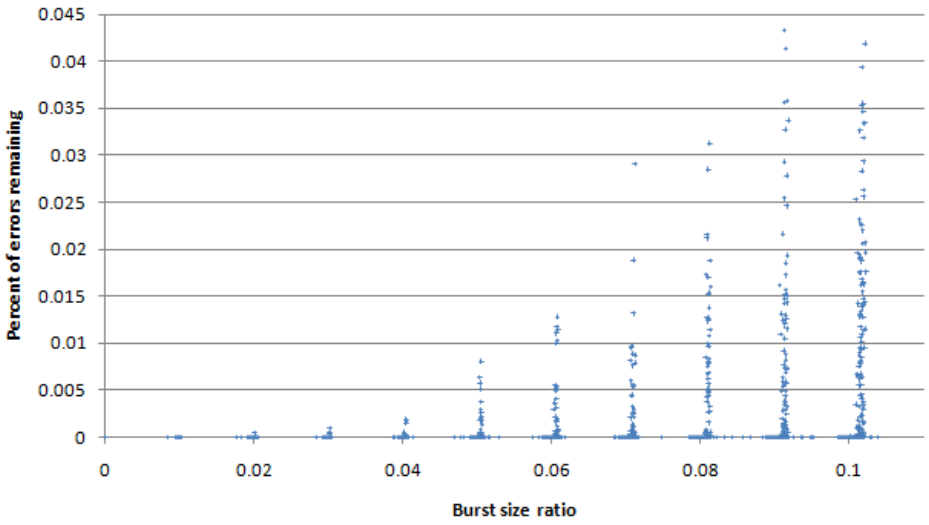


Figure 20. Failure of *Winnow* with single burst errors, no initial permutation

5.1. Recommendations for Future Research

The experiments designed in this research admit limitations, and each of these limitations can be used to identify areas for future research to further improve *Winnow* and characterize its strengths and weaknesses. The following list identifies areas for improving this work with regards to studying *Winnow*.

1. ***Improving the variant used in Experiment 2***

   Experiment 2 admits that there is a shortcoming in using error rate estimates for selecting the best next block size, and addresses this issue by employing a naïve mechanism for estimating the number of errors expected to be corrected and introduced in the current pass. This estimation mechanism is currently only used for a block size of 8-bits for simplicity, and is a *stub* in the author's current implementation of *Winnow* for other block sizes. This estimation

mechanism can be generalized to include other block sizes, which would likely improve the overall throughput of the *Winnow* protocol.

2. ***A study of a wider selection of block sizes***

   In the seminal research on *Winnow*, Buttler et al. use block sizes of 8 bits through 256 bits. This research uses block sizes of 8 bits through 1024 bits, the idea being that a larger block would be more suitable because less parity bits would be exchanged for very low error rates. It would be interesting to determine the point at which the block size becomes too large, that is, the number of bits sent in syndromes is larger than the number of unnecessary parity bits sent. This information can be used to bound the maximum block size. Indeed, a bound is needed on the maximum block size, because if the block size exceeds the number of bits in the sifted key *Winnow* will not function properly.

3. ***Verification of definition of "ideal block size"***

   This author's definition of *ideal block size* assumes that the ideal block contains exactly one error. However, the ideal block may in fact contain less or more than zero errors on average. For example, the authors of *Cascade* experimentally determined that the ideal block contains approximately 0.69 errors on average, while intuitively one might reason that exactly one error per block might be ideal. An experimental characterization of the definition of *ideal block size* for *Winnow* would be of interest.

4. **C++ *implementation improvements***

   Winnow is highly conducive to multithreading, as most of its operations can be done in parallel. The author's implementation does not exploit this property, and a parallelizable implementation would likely improve the speed of *Winnow* dramatically. Also, the author's implementation of matrix multiplication for producing syndromes is not as fast as it could be. A speed increase could be seen by computing the parity of each row of the parity-check matrix *logical AND'ed* with each block of sifted key.

5. ***Comparison with other error reconciliation protocols***

   Winnow is one protocol of many designed to address the issue of error reconciliation in QKD. A detailed comparison of *Winnow* with these protocols, for example Turbo Codes and Low Density Parity Check Codes, would be valuable in determining the benefits and weaknesses of each.

Appendix A: Sample BB84 Key Distribution

Quantum Channel																																	
Alice: Random bits for key material --	1	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	1	0	1	1	1	1	0	1	0	1	0	0	0	1
Alice: Random basis choice (0 is rectilinear) --	0	1	0	1	1	1	0	0	1	0	1	0	1	1	1	0	1	0	1	0	0	1	0	0	1	1	1	0	1	0	0	1	0
Alice: Polarization of qubit sent --	↕	↖	↔	↖	↗	↖	↔	↕	↗	↔	↗	↕	↗	↖	↗	↕	↗	↕	↗	↕	↔	↖	↕	↕	↖	↗	↖	↔	↗	↔	↔	↗	↕
Bob: Random measurement basis choice --	0	1	0	1	0	1	1	1	0	0	1	0	1	1	1	0	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1	1	1
Bob: Bit received --	1	-	0	1	-	1	1	1	1	0	0	-	0	1	0	-	0	0	1	-	0	1	1	0	1	-	0	0	1	-	-	0	0
Public Discussion																																	
Bob: Announces basis choices --	0		0	1		1	1	1	0	0	1		1	1	1		0	0	0		1	0	1	1	1		0	0	0			1	1
Alice: Confirms correct choices --	√		√	√		√				√	√		√	√	√			√							√			√				√	
Alice and Bob: Obtain sifted key --	1		0	1		1				0	0		0	1	0			0							1			0				0	
Bob: Reveal random subset of bits --			0							0								0							1								
Alice: Confirms bits, or identifies errors --			ok							ok								ok							ok							ok	
Remaining shared secret --	1			1		1					0		0	1	0													0					

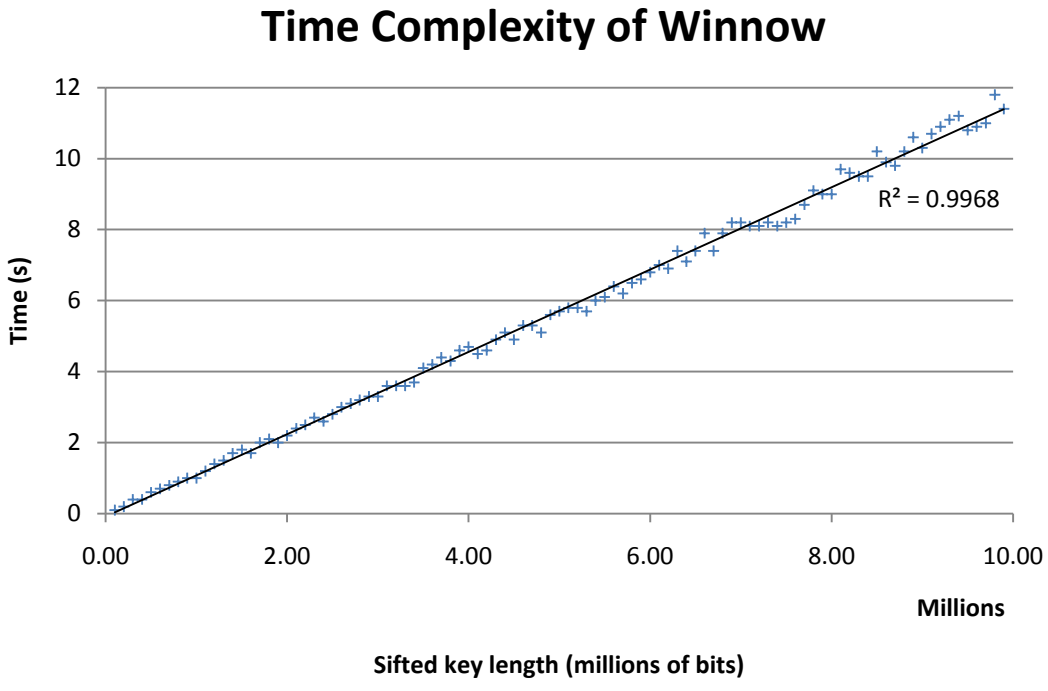
LEGEND

↔ 0-degree polarization	- Bob did not receive a bit
↕ 90-degree polarization	√ Basis matches
↗ 45-degree polarization	ok Bit is not in error
↖ 135-degree polarization	

Appendix B: Additional Analyses

Time Complexity

The time it takes for Winnow to completely eliminate all errors appears to scale linearly with sifted key length. The error rate is, of course, also a factor as the error rate determines the number of passes Winnow requires to correct all errors. To test the complexity, Winnow was performed on a sifted key with error rates of 1%-20% and sifted key length of 100,000 bits through 10,000,000 bits. For each percentage and length combination, an average run time was calculated in seconds from 10 trials. The graph below shows the averages of times for all twenty error rates and each sifted key length. The data showing each individual error rate for each sifted key length can be found on the accompanying CD along with the data used to generate the graph below.

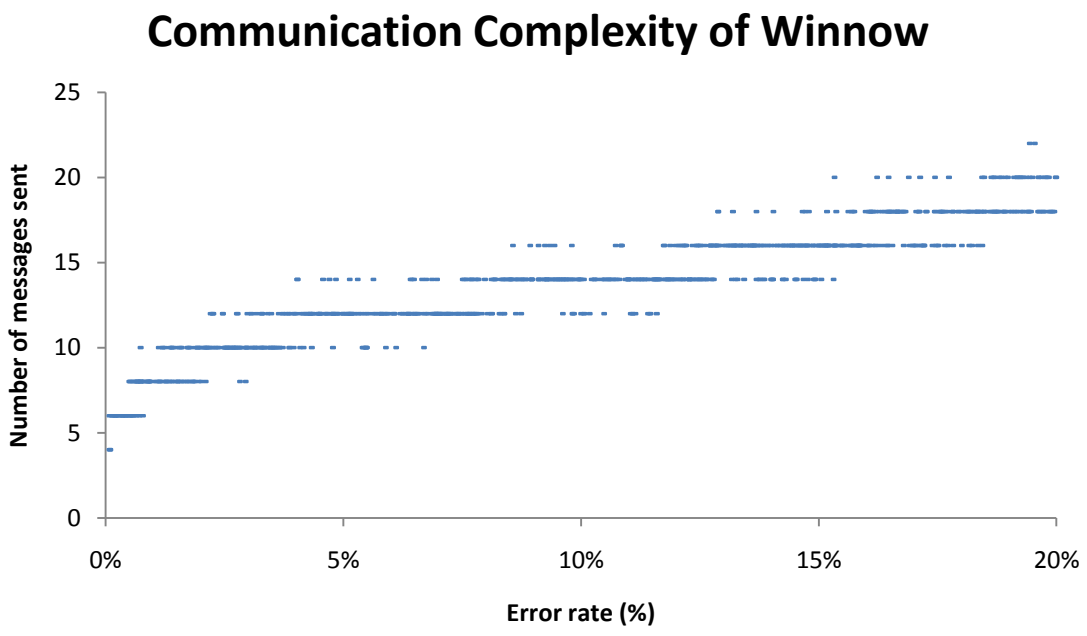


Communication Complexity

Communication complexity of a reconciliation protocol for use in a quantum key system can be defined in two ways:

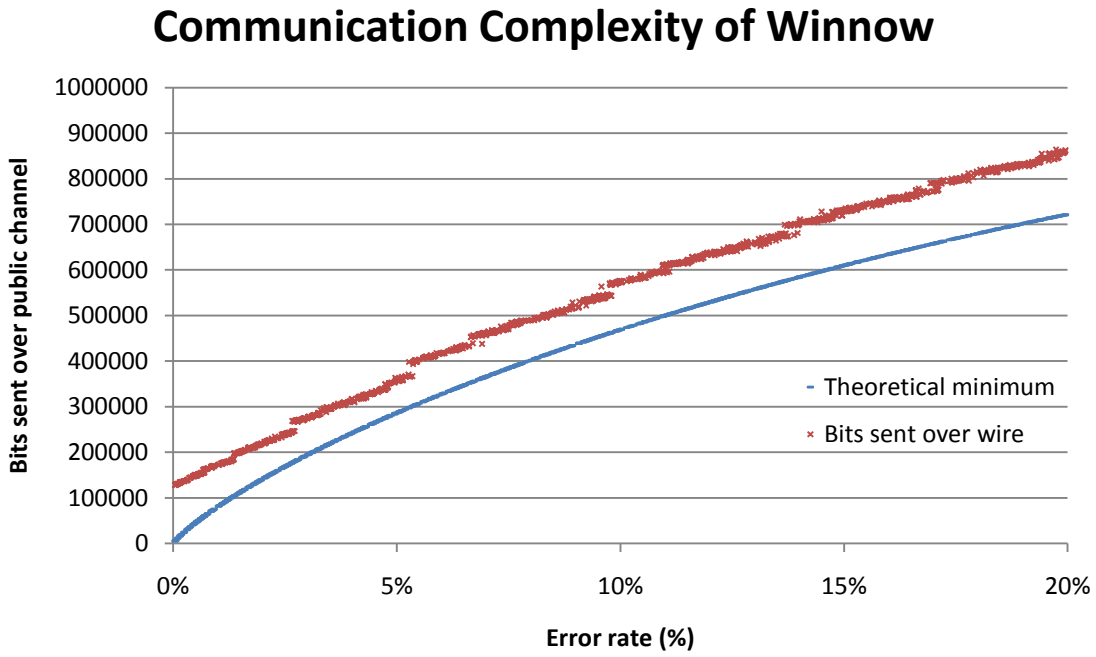
- The number of messages passed between Alice and Bob
- The number of bits sent between Alice and Bob

Using the first definition, the number of messages passed between Alice and Bob increases linearly with error rate, if *Winnow* is performed using the block sizes determined in Experiment 2 in this research (see Table 3). This is due to the fact that higher error rates demand a higher number of passes, and each pass requires exactly two messages to be sent between Alice and Bob. A comparison is shown below between the error rate and number of messages sent between Alice and Bob.



Using the second definition, the number of bits sent between Alice and Bob, we can draw a comparison between this curve and the theoretical minimum determined in

(Kollmitzer & Pivk, 2010). The number of bits sent follows a nice curve, as shown below. Note that the number of bits sent with respect to the size of the input is not interesting, as these numbers will most certainly be proportional. It is the effect of the error rate that is most interesting.



The data for both communication complexity graphs was taken from the data collected for Experiment 2. Note that the number of bits sent over the wire is the number of bits exposed to an eavesdropper.

Appendix C: Bit Buffer Class Code

```
/*
 * BitBuffer.h
 *
 *     Created on: Mar 28, 2011
 *                 Author: Kevin Lustic
 *
 *  This code can be found on the accompanying disc as well.
 */


#ifndef BITBUFFER_H_
#define BITBUFFER_H_

//#define using_bitmask


class BitBuffer {
public:
	BitBuffer();
	BitBuffer(unsigned int); // constructor
	virtual ~BitBuffer();

	unsigned int getBit(unsigned int);
	unsigned int getParity(unsigned int, unsigned int);
	unsigned int getLength();
	unsigned int getSeed();

	void setBit(unsigned int);
	void clearBit(unsigned int);
	void flipBit(unsigned int);
	void setLength(unsigned int);
	void setSeed(unsigned int);

	void printBuffer();
	void removeBit(unsigned int);
	void permuteBuffer();
	void newPermutation();

private:
	unsigned int *_buffer;
	unsigned int *_length;
	unsigned int *_seedVal;

	void ClearBuffer(void);

	#ifdef using_bitmask
	unsigned int *_bitMask;
	void InitializeBitMask(void);
	#endif
};

#endif
```

```
/*
 * BitBuffer.cpp
 *
 *     Created on: Mar 28, 2011
 *                 Author: Kevin Lustic
 *
 *  This code can be found on the accompanying disc as well.
 */


#define using_shifts             // Use shifts for bit manipulation
//#define using_bitmask          // Use bitmasks for bit manipulation
#define enable_debug_mode // Enable comments such as bounds checking

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "BitBuffer.h"

#define BITS_PER_WORD  32 // Set bits per word according to architecture (32-
or 64-bit)
#define BYTES_PER_WORD  (unsigned int)(BITS_PER_WORD/8)
#define MAX_BUFFER_ELEMENTS  4096
#define MAX_BITS  (MAX_BUFFER_ELEMENTS*BITS_PER_WORD)

BitBuffer::BitBuffer() {
       printf("You've used the default constructor. You should use the other
constructor.\n");
}

BitBuffer::  BitBuffer(unsigned int size) {
       _length = new unsigned int(size);

       srand(time(NULL));
       _seedVal = new unsigned int(0); //random seed for permutation

       _buffer = new unsigned int[*_length/BITS_PER_WORD+1];
       ClearBuffer();

       #ifdef using_bitmask
       _bitMask = new unsigned int [64];
       InitializeBitMask();
       #endif
}

BitBuffer::~BitBuffer() {
       delete _length;
       delete _buffer;
       delete _seedVal;

       #ifdef using_bitmask
       delete _bitMask;
       #endif
```

```
}

unsigned int BitBuffer::getLength() {
	return *_length;
}

void BitBuffer::setLength(unsigned int size) {
	_length = new unsigned int (size);
}

unsigned int BitBuffer::getSeed() {
	return *_seedVal;
}

void BitBuffer::setSeed(unsigned int seed) {
	_seedVal = new unsigned int (seed);
}


//////////////////////////// Start methodology-dependent functions
////////////////////////////////

#ifdef using_shifts

unsigned int BitBuffer::getBit(unsigned int index) {
	#ifdef enable_debug_mode
	if(index >= *_length) {
		printf("In getBit: index out of bounds. Returning 0.\n");
		return 0;
	}
	#endif

	return (_buffer[index/BITS_PER_WORD]>>(index%BITS_PER_WORD))&0x1;
}

void BitBuffer::setBit(unsigned int index) {
	#ifdef enable_debug_mode
	if(index >= *_length) {
		printf("In setBit: index out of bounds. Returning 0.\n");
		return;
	}
	#endif

	_buffer[index/BITS_PER_WORD] |= 1<<(index%BITS_PER_WORD);

	return;
}

void BitBuffer::clearBit(unsigned int index) {
	#ifdef enable_debug_mode
	if(index >= *_length) {
		printf("In clearBit: index out of bounds. Returning 0.\n");
		return;
```

```
	}
	#endif

	_buffer[index/BITS_PER_WORD] &= ~(1 << (index%BITS_PER_WORD));
	return;
}

void BitBuffer::flipBit(unsigned int index) {
	#ifdef enable_debug_mode
	if(index >= *_length) {
		printf("In clearBit: index out of bounds. Returning 0.\n");
		return;
	}
	#endif

	_buffer[index/BITS_PER_WORD] ^= 1<<(index%BITS_PER_WORD);

	return;
}

#endif // for using bit shifts for manipulation

#ifdef using_bitmask

unsigned int BitBuffer::getBit(unsigned int index) {
	#ifdef enable_debug_mode
	if(index >= *_length) {
		printf("In getBit: index out of bounds. Returning 0.\n");
		return 0;
	}
	#endif

	unsigned int Element = index / BITS_PER_WORD;
	unsigned int Bit = index % BITS_PER_WORD;

	if(_buffer[Element]&_bitMask[Bit])
		return 1;

	else
		return 0;
}

void BitBuffer::setBit(unsigned int index) {
	#ifdef enable_debug_mode
	if(index >= *_length) {
		printf("In setBit: index out of bounds. Returning 0.\n");
		return;
	}
	#endif

	unsigned int Element = index / BITS_PER_WORD;
	unsigned int Bit = index % BITS_PER_WORD;
```

```
	_buffer[Element] |= _bitMask[Bit];

	return;
}

void BitBuffer::clearBit(unsigned int index) {
	#ifdef enable_debug_mode
	if(index >= *_length) {
		printf("In clearBit: index out of bounds. Returning 0.\n");
		return;
	}
	#endif

	unsigned int Element = index / BITS_PER_WORD;
	unsigned int Bit = index % BITS_PER_WORD;

	_buffer[Element] &= (~_bitMask[Bit]);

	return;
}

void BitBuffer::flipBit(unsigned int index) {
	#ifdef enable_debug_mode
	if(index >= *_length) {
		printf("In flipBit: index out of bounds. Returning 0.\n");
		return;
	}
	#endif

	unsigned int Element = index / BITS_PER_WORD;
	unsigned int Bit = index % BITS_PER_WORD;

	if(_buffer[Element] & _bitMask[Bit])
		_buffer[Element] &= (~_bitMask[Bit]);

	else
		_buffer[Element] |= _bitMask[Bit];

	return;
}

#endif

////////////////////////////// End methodology-dependent functions
//////////////////////////////

/*
 * getParity(start,end)
 */
unsigned int BitBuffer::getParity(unsigned int start, unsigned int finish) {
	#ifdef enable_debug_mode
	if(start >= *_length || finish >= *_length || finish<start) {
		printf("In getParity: improper bounds. Returning 0.\n");
```

```
		printf("start/end is %d/%d\n",start,finish);

		return 0;
	}
	#endif

	unsigned int parity=0;
	unsigned int startInt=0, endInt=0;
	unsigned int startIndex = start/BITS_PER_WORD;
	unsigned int endIndex = finish/BITS_PER_WORD;

	// set all bits
	unsigned int startMask=~0;
	unsigned int endMask=~0;

	endMask >>= ((endIndex+1)*BITS_PER_WORD-finish-1);
	endInt = _buffer[endIndex] & endMask;

	startMask <<= (start-(startIndex)*BITS_PER_WORD);
	startInt = _buffer[startIndex] & startMask;

	if(startIndex == endIndex) {
		startMask &= endMask;
		parity ^= _buffer[startIndex] & startMask;
	}

	else {
		parity ^= startInt;
		for(unsigned int i=startIndex+1; i<endIndex; i++)
			parity ^= _buffer[i];
		parity ^= endInt;
	}

	for(unsigned int i=BITS_PER_WORD/2; i>=1; i/=2) {
		parity = parity ^ (parity >> i);
	}

	return parity & 0x1;
}

// Bits located at index i in _permutation will be switched to index
_permutation[i]
void BitBuffer::permuteBuffer() {
	srand(*_seedVal);
	unsigned int oldIndex = 0;
	unsigned int newIndex = 0;
	unsigned int tempNew,tempOld;
	unsigned int mask1=0, mask2=0;
	unsigned int length = *_length;

	// construct permuted bit buffer
	for(unsigned int counter=0; counter < length; counter++) {
```

```
			oldIndex = counter;
			newIndex = (unsigned int)((double)rand()/RAND_MAX * length);

			mask1 = ~0 ^ (1 << (newIndex%BITS_PER_WORD));
			tempNew = _buffer[newIndex/BITS_PER_WORD] & mask1;
			mask1 = ~mask1;
			tempNew |= (_buffer[oldIndex/BITS_PER_WORD] & mask1);

			mask2 = ~0 ^ (1 << (oldIndex%BITS_PER_WORD));
			tempOld = _buffer[oldIndex/BITS_PER_WORD] & mask2;
			mask2 = ~mask2;
			tempOld |= (_buffer[newIndex/BITS_PER_WORD] & mask2);

			_buffer[newIndex/BITS_PER_WORD] = tempNew;
			_buffer[oldIndex/BITS_PER_WORD] = tempOld;
		}

		return;
}

void BitBuffer::printBuffer() {
		#ifdef enable_debug_mode
		if(*_length > 100) {
			printf("Buffer larger than 100 bits, you probably don't want this
printed.\n");
			return;
		}
		#endif

		for(unsigned int i=0; i < *_length; i++) {
			printf("%d", getBit(i));
		}
		printf("\n");

		return;
}

void BitBuffer::ClearBuffer(void) {
		for(unsigned int i=0; i<*_length/BITS_PER_WORD; i++)
			_buffer[i]=0;
}

#ifdef using_bitmask

void BitBuffer::InitializeBitMask(void) {
#if BYTES_PER_WORD==4
		_bitMask[0]= 0x00000001;
		_bitMask[1]= 0x00000002;
		_bitMask[2]= 0x00000004;
		_bitMask[3]= 0x00000008;
		_bitMask[4]= 0x00000010;
		_bitMask[5]= 0x00000020;
		_bitMask[6]= 0x00000040;
```

```
        _bitMask[7]= 0x00000080;
        _bitMask[8]= 0x00000100;
        _bitMask[9]= 0x00000200;
        _bitMask[10]=0x00000400;
        _bitMask[11]=0x00000800;
        _bitMask[12]=0x00001000;
        _bitMask[13]=0x00002000;
        _bitMask[14]=0x00004000;
        _bitMask[15]=0x00008000;
        _bitMask[16]=0x00010000;
        _bitMask[17]=0x00020000;
        _bitMask[18]=0x00040000;
        _bitMask[19]=0x00080000;
        _bitMask[20]=0x00100000;
        _bitMask[21]=0x00200000;
        _bitMask[22]=0x00400000;
        _bitMask[23]=0x00800000;
        _bitMask[24]=0x01000000;
        _bitMask[25]=0x02000000;
        _bitMask[26]=0x04000000;
        _bitMask[27]=0x08000000;
        _bitMask[28]=0x10000000;
        _bitMask[29]=0x20000000;
        _bitMask[30]=0x40000000;
        _bitMask[31]=0x80000000;
#elif BYTES_PER_WORD == 8
        _bitMask[0]= 0x0000000000000001;
        _bitMask[1]= 0x0000000000000002;
        _bitMask[2]= 0x0000000000000004;
        _bitMask[3]= 0x0000000000000008;
        _bitMask[4]= 0x0000000000000010;
        _bitMask[5]= 0x0000000000000020;
        _bitMask[6]= 0x0000000000000040;
        _bitMask[7]= 0x0000000000000080;
        _bitMask[8]= 0x0000000000000100;
        _bitMask[9]= 0x0000000000000200;
        _bitMask[10]=0x0000000000000400;
        _bitMask[11]=0x0000000000000800;
        _bitMask[12]=0x0000000000001000;
        _bitMask[13]=0x0000000000002000;
        _bitMask[14]=0x0000000000004000;
        _bitMask[15]=0x0000000000008000;
        _bitMask[16]=0x0000000000010000;
        _bitMask[17]=0x0000000000020000;
        _bitMask[18]=0x0000000000040000;
        _bitMask[19]=0x0000000000080000;
        _bitMask[20]=0x0000000000100000;
        _bitMask[21]=0x0000000000200000;
        _bitMask[22]=0x0000000000400000;
        _bitMask[23]=0x0000000000800000;
        _bitMask[24]=0x0000000001000000;
        _bitMask[25]=0x0000000002000000;
        _bitMask[26]=0x0000000004000000;
```

```
        _bitMask[27]=0x0000000008000000;
        _bitMask[28]=0x0000000010000000;
        _bitMask[29]=0x0000000020000000;
        _bitMask[30]=0x0000000040000000;
        _bitMask[31]=0x0000000080000000;
        _bitMask[32]=0x0000000100000000;
        _bitMask[33]=0x0000000200000000;
        _bitMask[34]=0x0000000400000000;
        _bitMask[35]=0x0000000800000000;
        _bitMask[36]=0x0000001000000000;
        _bitMask[37]=0x0000002000000000;
        _bitMask[38]=0x0000004000000000;
        _bitMask[39]=0x0000008000000000;
        _bitMask[40]=0x0000010000000000;
        _bitMask[41]=0x0000020000000000;
        _bitMask[42]=0x0000040000000000;
        _bitMask[43]=0x0000080000000000;
        _bitMask[44]=0x0000100000000000;
        _bitMask[45]=0x0000200000000000;
        _bitMask[46]=0x0000400000000000;
        _bitMask[47]=0x0000800000000000;
        _bitMask[48]=0x0001000000000000;
        _bitMask[49]=0x0002000000000000;
        _bitMask[50]=0x0004000000000000;
        _bitMask[51]=0x0008000000000000;
        _bitMask[52]=0x0010000000000000;
        _bitMask[53]=0x0020000000000000;
        _bitMask[54]=0x0040000000000000;
        _bitMask[55]=0x0080000000000000;
        _bitMask[56]=0x0100000000000000;
        _bitMask[57]=0x0200000000000000;
        _bitMask[58]=0x0400000000000000;
        _bitMask[59]=0x0800000000000000;
        _bitMask[60]=0x1000000000000000;
        _bitMask[61]=0x2000000000000000;
        _bitMask[62]=0x4000000000000000;
        _bitMask[63]=0x8000000000000000;
#endif
}

#endif
```

Appendix D: Winnow Class Code

```
/*
 * Kevin Lustic
 * Air Force Institute of Technology
 * Winnow.cpp
 *
 *  Created on: Mar 18, 2011
 *  Last Updated: Mar 30, 2011
 *
 *  This code can be found on the accompanying disc as well.
 */

#ifndef WINNOW_H_
#define WINNOW_H_

#include "BitBuffer.h"

class Winnow {
public:
	Winnow();
	Winnow(BitBuffer *, unsigned int);
	virtual ~Winnow();

	int firstPass();
	int nextPass();
	double setBlockSchedule(unsigned int);

	// Parity operation methods
	void getParities(BitBuffer *);
	void discardParityBits();
	unsigned int disagreeingBlockParities(BitBuffer *, BitBuffer *, int *);

	// Syndrome operation methods
	void printMatrix();
	void fixWithSyndrome(unsigned int, unsigned int);
	void discardSyndromeBits (unsigned int);
	void discardSyndromeBits (int *, unsigned int);
	unsigned int getSyndrome(unsigned int);

	// Misc getter methods
	void getCurrentKey(BitBuffer *);
	void getBlockSizeSchedule(unsigned int *);
	unsigned int getCurrentKeyLength();
	unsigned int getBitsExposed();
	unsigned int getNetBitsExposed();
	unsigned int numRemainingErrors(BitBuffer *);
	unsigned int getNumRemainingPasses();

	// BitBuffer wrappers
	void setSeedValue(unsigned int);
	unsigned int getSeedValue();

private:
```

```
        bool _setterLocked;
        BitBuffer *_keyString;
        unsigned int _blockSize;
        unsigned int _numOfBlocks;
        unsigned int _bitsExposed;
        unsigned int _netBitsExposed;
        unsigned int _syndromeLength;
        unsigned int *_blockSizeSchedule;
        unsigned int **_parityCheckMatrix;

        void createMatrix();
};

#endif /* WINNOW_H_ */
```

```
/*
 * Kevin Lustic
 * Air Force Institute of Technology
 * Winnow.cpp
 *
 *  Created on: Mar 18, 2011
 *  Last Updated: May 18, 2011
 *
 *  This code can be found on the accompanying disc as well.
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "Winnow.h"

#define permute_bits_before_first_pass
#define permute_bits_between_passes


Winnow::Winnow() {
	printf("Default constructor for Winnow... use other constructor.\n");
}

/*
 * The constructor
 *
 * Input: Sifted key
 *
 * - Initializes the Winnow class's member bit buffer
 * - Sets block size to 8 bits, for initial error estimation
 * - Sets bits exposed and net bits exposed to 0. These track Eve's info gain
from Winnowing *
 */
Winnow::Winnow(BitBuffer *rawKey, unsigned int permSeed) {
	_keyString = rawKey;
	_keyString->setSeed(permSeed);

	_bitsExposed = 0;
	_netBitsExposed = 0;
	_setterLocked = false;

//	_blockSizeSchedule = new unsigned int[8];

}

Winnow::~Winnow() {}

/*
 * Winnow::firstPass
 *
 * Prepares for the first pass. This includes initializing the block size
schedule,
```

```
 * determining the initial blocksize, syndrome length, and number of blocks,
and
 * generating the parity-check matrix.
 */
int Winnow::firstPass() {

	// For the first pass, we will always choose a block size of 8
	_syndromeLength = 3;
	_blockSize = 8;

	// Note that this number does not include an incomplete final block
	_numOfBlocks = _keyString->getLength()/_blockSize;

	// Generation of the parity-check matrix
	createMatrix();

	// Permute the bits in the key buffer
	#ifdef permute_bits_before_first_pass
	_keyString->permuteBuffer();
	#endif

	return 0;
}

/*
 * Winnow::nextPass
 *
 * Before the next pass of Winnow is performed this function is called. This
function
 * adjusts the blocksize according the schedule, creates the new parity check
matrix and
 * adjusts any other values related to the block size.
 */
int Winnow::nextPass() {

	for(int i=0; i<8; i++) {
		if(_blockSizeSchedule[i] > 0) {
			_syndromeLength = (unsigned int)(i+3);
			_blockSize = 1<<_syndromeLength;
			_blockSizeSchedule[i]--;
			break;
		}
		else if (i>=7) {
			// We reached end of schedule with no block choices left
			printf("Time to terminate.\n");
			return (-1);
		}
	}

	_numOfBlocks = _keyString->getLength()/_blockSize;
	createMatrix();

	// Permute the bits in the key buffer
```

```
	#ifdef permute_bits_between_passes
	_keyString->permuteBuffer();
	#endif

	return 0;
}


/*
 * Winnow::getCurrentKeyLength()
 *
 * Output: Length of the current key.
 */
unsigned int Winnow::getCurrentKeyLength() {
	return _keyString->getLength();
}


/*
 * Winnow::getNumRemainingPasses
 *
 * Output: Number of passes left
 *
 * Returns the number of passes left according to block schedule. This
function
 * is responsible for the termination of the algorithm.
 */
unsigned int Winnow::getNumRemainingPasses() {
	unsigned int count=0;

	for(int i=0; i<8; i++)
		count += _blockSizeSchedule[i];

	return count;
}

/*
 * Winnow::getParities
 *
 * Input:  BitBuffer *
 * Output: void
 *
 * Gets the parities for each of the blocks, and stores them in the BitBuffer
 * that is passed by reference. The indices of the bit buffer correspond to
the
 * block number of the parity bit stored there.
 */
void Winnow::getParities(BitBuffer *parityBuffer) {
	parityBuffer->setLength(_numOfBlocks);

	unsigned int start=0, end=0;
	unsigned int parity=0;
```

```
	// Iterate through the blocks. Note we ignore the last block if it is
not a full block.
	for(unsigned int i=0; i < _numOfBlocks; i++) {
		start = i*_blockSize;
		end = start+_blockSize-1;

		parity = _keyString->getParity(start,end); // Parity of bits from
start->end, inclusive

		// Each parity bit is counted as a bit exposed
		_bitsExposed++;
		_netBitsExposed++;

		// Update our parity buffer
		if(parity==1)
			parityBuffer->setBit(i);
		else {
			parityBuffer->clearBit(i);
		}
	}

	return;
}

/*
 * Winnow::disagreeingBlockParities
 *
 * Input:  Alice's parity buffer, Bob's parity buffer, array of integers
 * Output: The number of blocks containing an odd number of errors
 *
 * Determines the block numbers of blocks where Alice and Bob's parities do
not match,
 * and store those block numbers in the array of integers (the third
argument).
 */
unsigned int Winnow::disagreeingBlockParities(BitBuffer *aliceParities,
BitBuffer *bobParities, int *mismatches) {
	unsigned int counter = 0;

	// Ensure we are passed valid buffers to compare
	if(aliceParities->getLength() != bobParities->getLength()) {
		printf("Incompatible buffers... cannot determine if they
disagree.\n");
	}

	else {
		for(unsigned int i=0; i < aliceParities->getLength(); i++) {

			// If Alice/Bob have parities that don't match, add the
index (block #) to our list
			if(aliceParities->getBit(i) != bobParities->getBit(i)) {
				mismatches[counter]=i;
				counter++;
```

```
                }
            }
        }

        return counter;
}


/*
 * Winnow::discardParityBits
 *
 * Input:  (void)
 * Output: (void)
 *
 * Discards a bit from each block as a part of the Winnow privacy maintenance.
This
 * implementation discards the first bit from each block, for ease of coding.
Discard
 * is performed by copying the bits of each block which aren't part of the key
string
 * backwards.
 *
 * Note that the final block is ignored if incomplete.
 */
void Winnow::discardParityBits() {
        unsigned int newIndex=0;

        for(unsigned int i=0; i<_keyString->getLength(); i++) {
                if(i%_blockSize == 0 && i != _numOfBlocks*_blockSize) {
                        // We want to remove this bit. Do not copy it.
                        _netBitsExposed--;
                }

                // If it isn't the first bit in the block, copy it.
                else {
                        if(_keyString->getBit(i)==1)
                                _keyString->setBit(newIndex);
                        else {
                                _keyString->clearBit(newIndex);
                        }
                        newIndex++;
                }
        }

        // New block size, since we deleted one bit from each
        _blockSize--;

        // New length of key, since we delete one bit per block
        _keyString->setLength(_keyString->getLength()-_numOfBlocks);

        return;
}
```

```
/*
 * Winnow::createMatrix
 *
 * Function for generating the parity check matrix that Alice uses for
computing
 * syndromes, and that Bob uses for correcting errors.
 */

void Winnow::createMatrix() {
	int blockSize = (1<<_syndromeLength)-1;

	_parityCheckMatrix = new unsigned int*[_syndromeLength];

	for(unsigned int i=0; i<_syndromeLength; i++) {
		_parityCheckMatrix[i] = new unsigned int[blockSize];
	}

	for(unsigned int i=0; i<_syndromeLength; i++) {
		for(int j=1; j<=blockSize; j++) {
			_parityCheckMatrix[i][j-1] = j/(1<<i) & 0x1;
		}
	}

	return;
}

/*
 * Winnow::getSyndrome
 *
 * Input:  The block number of the block for which to calculate the syndrome
 * Output: The syndrome, in unsigned int form
 *
 * The syndrome is returned as an unsigned integer. The blocksize would have
to
 * be 2^32-1 to break this method, which is an unrealistic block size in
practice.
 * Unsigned integers for this purpose is less unwieldy than, say, a separate
buffer
 * for every syndrome.
 *
 * An example is a syndrome of '1 1 0' would be returned as '6' rather than a
 * three-element buffer.
 */
unsigned int Winnow::getSyndrome(unsigned int blockNumber) {
	// Ensure the block number is legal
	if(blockNumber > _numOfBlocks) {
		printf("Illegal block number. Returning blockSize+1 for new
syndrome.\n");
		return (_blockSize+1);
	}

	unsigned int temp=0;
	unsigned int newSyndrome=0;
```

```
	// Compute the highest order bit of the syndrome first and work down
	for(int i=_syndromeLength-1; i>=0; i--) {
		newSyndrome <<= 1; // Push previous bit up

		// Multiply the block by the (i-1)th row of the parity check
matrix
		for(unsigned int j=0; j<_blockSize; j++) {
			temp = (temp + _parityCheckMatrix[i][j]*_keyString-
>getBit(blockNumber*_blockSize+j)) & 0x1;
		}

		newSyndrome += temp; // Add the resulting sum to the syndrome

		temp=0;
	}

	// Number of bits exposed is equal to the number of bits in the syndrome
	_bitsExposed+=_syndromeLength;
	_netBitsExposed+=_syndromeLength;

	return newSyndrome;
}

/*
 * Winnow::fixWithSyndrome
 *
 * Input: the number of the block containing the error, Alice's syndrome of
that block
 *
 * This is used by Bob. Bob computes the syndrome of his block, and xor's it
with the
 * syndrome from Alice. The result is an offset pointing to the exact location
of the
 * erroneous bit, if the first bit of the block is considered to be at
position '1'.
 */
void Winnow::fixWithSyndrome(unsigned int blockNumber, unsigned int syndrome)
{
	// Get the syndrome of the corresponding block
	unsigned int mySyndrome = this->getSyndrome(blockNumber);

	// Store in syndromesyndrome the result of xor-ing the two syndromes.
This will
	// give an offset representing the location of the error to fix
	syndrome ^= mySyndrome;

	if(syndrome==0) { /* The error was discarded in the parity cleanup! */ }

	else
		_keyString->flipBit(blockNumber*_blockSize+(syndrome-1));

	return;
```

```
}

/*
 * Winnow::discardSyndromeBits
 *
 * Input: block numbers of the blocks for which to discard the bits, number of
said blocks
 *
 * The bits at indices of the form 2^j-1 are removed. These correspond to the
 * linearly independent columns of the parity check matrix. Note that this
 * appears to be O(n^2) but the inner for loop is deceiving. This operation
 * is just O(n).
 */
void Winnow::discardSyndromeBits(int *errorBlocks, unsigned int length) {
	unsigned int errorBlocksCounter=0,	newCounter=0,oldCounter=0;
	for(unsigned int i=0; i<_numOfBlocks; i++) {
		if(errorBlocksCounter<length && errorBlocks[errorBlocksCounter]
== (int)i) {
			errorBlocksCounter++;
			unsigned int power=0;

			// Iterate through the block
			for(unsigned int j=0; j<_blockSize; j++) {

				if(j+1 == (unsigned int)(1<<power)) {
					_netBitsExposed--;
					power++;
					oldCounter++;
				}

				else {
					if(_keyString->getBit(oldCounter))
						_keyString->setBit(newCounter);
					else {
						_keyString->clearBit(newCounter);
					}

					newCounter++;
					oldCounter++;
				}
			} // end for
		} //end if

		else {
			if(_keyString->getBit(oldCounter))
				_keyString->setBit(newCounter);
			else {
				_keyString->clearBit(newCounter);
			}
			newCounter++;
			oldCounter++;
		} //end else
	} //end for
```

```
	while (oldCounter < _keyString->getLength()) {
		if(_keyString->getBit(oldCounter))
			_keyString->setBit(newCounter);
		else {
			_keyString->clearBit(newCounter);
		}
		newCounter++;
		oldCounter++;
	}

	_keyString->setLength(newCounter);

	return;
}



/*
 * Winnow::getCurrentKey
 *
 * Input: BitBuffer pointer
 *
 * Copies the current key stored in _keyString into myBuf
 */
void Winnow::getCurrentKey(BitBuffer *myBuf) {
	myBuf = _keyString;
	return;
}

/*
 * Winnow::getBitsExposed
 *
 * Returns the gross number of bits exposed.
 */
unsigned int Winnow::getBitsExposed() {
	return _bitsExposed;
}

/*
 * Winnow::getNetBitsExposed
 *
 * Returns the net number of bits exposed (gross number - bits discarded)
 */
unsigned int Winnow::getNetBitsExposed() {
	return _netBitsExposed;
}


/*
 * Winnow::numRemainingErrors
 *
 * Input:  BitBuffer containing the key to compare to the member keystring
```

```
 * Output: number of places where the two strings do not match
 *
 * This is only used for statistics collection and verification of algorithm
 * effectiveness; it won't be used in practice.
 */
unsigned int Winnow::numRemainingErrors(BitBuffer *otherBuffer) {
	unsigned int counter=0;

	for(unsigned int i=0; i<_keyString->getLength(); i++) {
		if(otherBuffer->getBit(i) != _keyString->getBit(i))
			counter++;
	}

	return counter;
}

/*
 * Winnow::printMatrix
 *
 * Used to display the parity check matrix, for debugging.
 */
void Winnow::printMatrix() {
	printf("Parity-Check matrix:\n");
	for(unsigned int i=0; i<_syndromeLength; i++) {
		for(unsigned int j=0; j<_blockSize; j++) {
			printf("%d ",_parityCheckMatrix[i][j]);
		}
		printf("\n");
	}
	return;
}

/*
 * BitBuffer wrappers
 *
 * These are accessor wrappers for determining and setting the seed used for
 * the random permutation of the bits.
 */

void Winnow::setSeedValue(unsigned int seed) {
	_keyString->setSeed(seed);
}

unsigned int Winnow::getSeedValue() {
	return _keyString->getSeed();
}


double Winnow::setBlockSchedule(unsigned int pe) {

	if(_setterLocked) {
		printf("Winnow::setBlockSchedule -> You shouldn't be accessing
this!\n");
```

```
	return .01;
}

_setterLocked = true;

long double newRate;
long double bl = (long double)_numOfBlocks;

newRate = (bl-2*pe)/bl;
newRate = .5L*(1-pow(newRate,.125L));

_blockSizeSchedule = new unsigned int[8];

if(newRate <= 0.01) {
	_blockSizeSchedule[0] = 1;
	_blockSizeSchedule[1] = 0;
	_blockSizeSchedule[2] = 0;
	_blockSizeSchedule[3] = 1;
	_blockSizeSchedule[4] = 0;
	_blockSizeSchedule[5] = 0;
	_blockSizeSchedule[6] = 1;
	_blockSizeSchedule[7] = 2;
}

else if(newRate <= 0.02) {
	_blockSizeSchedule[0] = 1;
	_blockSizeSchedule[1] = 0;
	_blockSizeSchedule[2] = 1;
	_blockSizeSchedule[3] = 0;
	_blockSizeSchedule[4] = 0;
	_blockSizeSchedule[5] = 1;
	_blockSizeSchedule[6] = 2;
	_blockSizeSchedule[7] = 1;
}

else if(newRate <= 0.03) {
	_blockSizeSchedule[0] = 1;
	_blockSizeSchedule[1] = 1;
	_blockSizeSchedule[2] = 0;
	_blockSizeSchedule[3] = 0;
	_blockSizeSchedule[4] = 1;
	_blockSizeSchedule[5] = 0;
	_blockSizeSchedule[6] = 2;
	_blockSizeSchedule[7] = 1;
}

else if(newRate <= 0.04) {
	_blockSizeSchedule[0] = 1;
	_blockSizeSchedule[1] = 1;
	_blockSizeSchedule[2] = 0;
	_blockSizeSchedule[3] = 0;
	_blockSizeSchedule[4] = 1;
	_blockSizeSchedule[5] = 0;
```

```
        _blockSizeSchedule[6] = 2;
        _blockSizeSchedule[7] = 1;
    }

    else if(newRate <= 0.05) {
        _blockSizeSchedule[0] = 1;
        _blockSizeSchedule[1] = 1;
        _blockSizeSchedule[2] = 0;
        _blockSizeSchedule[3] = 1;
        _blockSizeSchedule[4] = 0;
        _blockSizeSchedule[5] = 1;
        _blockSizeSchedule[6] = 2;
        _blockSizeSchedule[7] = 1;
    }

    else if(newRate <= 0.06) {
        _blockSizeSchedule[0] = 1;
        _blockSizeSchedule[1] = 1;
        _blockSizeSchedule[2] = 1;
        _blockSizeSchedule[3] = 0;
        _blockSizeSchedule[4] = 1;
        _blockSizeSchedule[5] = 0;
        _blockSizeSchedule[6] = 2;
        _blockSizeSchedule[7] = 2;
    }

    else if(newRate <= 0.07) {
        _blockSizeSchedule[0] = 2;
        _blockSizeSchedule[1] = 0;
        _blockSizeSchedule[2] = 1;
        _blockSizeSchedule[3] = 0;
        _blockSizeSchedule[4] = 1;
        _blockSizeSchedule[5] = 0;
        _blockSizeSchedule[6] = 2;
        _blockSizeSchedule[7] = 1;
    }

    else if(newRate <= 0.08) {
        _blockSizeSchedule[0] = 2;
        _blockSizeSchedule[1] = 1;
        _blockSizeSchedule[2] = 0;
        _blockSizeSchedule[3] = 0;
        _blockSizeSchedule[4] = 1;
        _blockSizeSchedule[5] = 0;
        _blockSizeSchedule[6] = 2;
        _blockSizeSchedule[7] = 1;
    }

    else if(newRate <= 0.09) {
        _blockSizeSchedule[0] = 2;
        _blockSizeSchedule[1] = 1;
        _blockSizeSchedule[2] = 0;
        _blockSizeSchedule[3] = 1;
```

```
        _blockSizeSchedule[4] = 0;
        _blockSizeSchedule[5] = 1;
        _blockSizeSchedule[6] = 2;
        _blockSizeSchedule[7] = 1;
}

else if(newRate <= 0.10) {
        _blockSizeSchedule[0] = 2;
        _blockSizeSchedule[1] = 1;
        _blockSizeSchedule[2] = 1;
        _blockSizeSchedule[3] = 0;
        _blockSizeSchedule[4] = 1;
        _blockSizeSchedule[5] = 0;
        _blockSizeSchedule[6] = 2;
        _blockSizeSchedule[7] = 1;
}

else if(newRate <= 0.11) {
        _blockSizeSchedule[0] = 3;
        _blockSizeSchedule[1] = 0;
        _blockSizeSchedule[2] = 1;
        _blockSizeSchedule[3] = 0;
        _blockSizeSchedule[4] = 1;
        _blockSizeSchedule[5] = 0;
        _blockSizeSchedule[6] = 2;
        _blockSizeSchedule[7] = 1;
}

else if(newRate <= 0.12) {
        _blockSizeSchedule[0] = 3;
        _blockSizeSchedule[1] = 1;
        _blockSizeSchedule[2] = 0;
        _blockSizeSchedule[3] = 0;
        _blockSizeSchedule[4] = 1;
        _blockSizeSchedule[5] = 0;
        _blockSizeSchedule[6] = 2;
        _blockSizeSchedule[7] = 1;
}

else if(newRate <= 0.13) {
        _blockSizeSchedule[0] = 3;
        _blockSizeSchedule[1] = 1;
        _blockSizeSchedule[2] = 0;
        _blockSizeSchedule[3] = 1;
        _blockSizeSchedule[4] = 0;
        _blockSizeSchedule[5] = 1;
        _blockSizeSchedule[6] = 2;
        _blockSizeSchedule[7] = 1;
}

else if(newRate <= 0.14) {
        _blockSizeSchedule[0] = 3;
        _blockSizeSchedule[1] = 1;
```

```
        _blockSizeSchedule[2] = 1;
        _blockSizeSchedule[3] = 0;
        _blockSizeSchedule[4] = 1;
        _blockSizeSchedule[5] = 0;
        _blockSizeSchedule[6] = 2;
        _blockSizeSchedule[7] = 1;
}

else if(newRate <= 0.15) {
        _blockSizeSchedule[0] = 4;
        _blockSizeSchedule[1] = 0;
        _blockSizeSchedule[2] = 1;
        _blockSizeSchedule[3] = 0;
        _blockSizeSchedule[4] = 1;
        _blockSizeSchedule[5] = 0;
        _blockSizeSchedule[6] = 2;
        _blockSizeSchedule[7] = 1;
}

else if(newRate <= 0.16) {
        _blockSizeSchedule[0] = 4;
        _blockSizeSchedule[1] = 1;
        _blockSizeSchedule[2] = 0;
        _blockSizeSchedule[3] = 1;
        _blockSizeSchedule[4] = 0;
        _blockSizeSchedule[5] = 0;
        _blockSizeSchedule[6] = 2;
        _blockSizeSchedule[7] = 1;
}

else if(newRate <= 0.17) {
        _blockSizeSchedule[0] = 4;
        _blockSizeSchedule[1] = 1;
        _blockSizeSchedule[2] = 1;
        _blockSizeSchedule[3] = 0;
        _blockSizeSchedule[4] = 1;
        _blockSizeSchedule[5] = 0;
        _blockSizeSchedule[6] = 2;
        _blockSizeSchedule[7] = 1;
}

else if(newRate <= 0.18) {
        _blockSizeSchedule[0] = 5;
        _blockSizeSchedule[1] = 0;
        _blockSizeSchedule[2] = 1;
        _blockSizeSchedule[3] = 0;
        _blockSizeSchedule[4] = 1;
        _blockSizeSchedule[5] = 0;
        _blockSizeSchedule[6] = 2;
        _blockSizeSchedule[7] = 1;
}

else if(newRate <= 0.19) {
```

```
		_blockSizeSchedule[0] = 5;
		_blockSizeSchedule[1] = 1;
		_blockSizeSchedule[2] = 0;
		_blockSizeSchedule[3] = 1;
		_blockSizeSchedule[4] = 0;
		_blockSizeSchedule[5] = 1;
		_blockSizeSchedule[6] = 2;
		_blockSizeSchedule[7] = 1;
	}

	else {
		_blockSizeSchedule[0] = 5;
		_blockSizeSchedule[1] = 1;
		_blockSizeSchedule[2] = 1;
		_blockSizeSchedule[3] = 0;
		_blockSizeSchedule[4] = 1;
		_blockSizeSchedule[5] = 0;
		_blockSizeSchedule[6] = 2;
		_blockSizeSchedule[7] = 2;
	}

	_blockSizeSchedule[0]--;

	return (double)newRate;
}

void Winnow::getBlockSizeSchedule(unsigned int *bss) {

	bss[0] = _blockSizeSchedule[0]+1;
	for(int i=1; i<8; i++)
		bss[i] = _blockSizeSchedule[i];

	return;
}
```

Bibliography

Ardehali, M., Chau, H. F., & Lo, H.-K. (1999, Jan 29). *arXiv:quant-ph/9803007v4*. Retrieved Mar 4, 2011

Bennett, C. H., & Brassard, G. (1984). Quantum Cryptography: Public Key Distribution and Coin Tossing. *International Conference on Computers, Systems & Signals Processing.* Bangalore, India.

Bennett, C. H., Bessette, F., Brassard, G., Salvail, L., & Smolin, J. (1992). Experimental quantum cryptography. *Journal of Cryptology , 5* (1), 3-28.

Bernstein, D. J., Buchmann, J., & Dahmen, E. (Eds.). (2010). *Post-Quantum Cryptography.* Berlin: Springer-Verlag Berlin Heidelberg.

Brassard, G. (1993). A bibliography of quantum cryptography. *ACM SIGACT News , 24* (3), 16-20.

Brassard, G., & Salvail, G. (1994). Secret-key reconciliation by public discussion. *Workshop on the theory and application of cryptographic techniques on Advances in cryptology* (pp. 410-423). Lofthus, Norway: Springer-Verlag.

Buttler, W. T., Lamoreaux, J. R., Torgerson, J. R., Nickel, G. H., Donahue, C. H., & Peterson, C. G. (2003). Fast, efficient error reconciliation for quantum cryptography. *Physical Review A , 67* (5).

Chen, K. (2000, Aug). Improvement of Reconciliation for Quantum Key Distribution. *Master's Thesis* . Rochester Institute of Technology.

Hankerson, D. R., Hoffman, D. G., Leonard, D. A., Lindner, C. G., Phelps, K. T., Rodger, C. A., et al. (1991). *Coding Theory and Cryptography: The Essentials.* New York: Marcel Dekker.

Kleinjung, T., & others. (2010, Feb 18). Retrieved Mar 13, 2011, from http://eprint.iacr.org/2010/006.pdf

Kollmitzer, C., & Pivk, M. (Eds.). (2010). Applied Quantum Cryptography. *Lecture Notes in Physics , 797 , 1*. Springer, Berlin Heidelberg.

Nakassis, A., Bienfang, J., & Williams, C. (2004). Expeditious reconciliation for practical quantum key distribution. *Quantum Information and Computation, Proc. SPIE 5436.*

Nguyen, K.-C. (2002). Extension des Protocoles de Réconciliation en Cryptographie Quantique. *Masters thesis* . Université Libre de Bruxelles.

Politi, A., Matthews, J. C., & O'Brien, J. L. (2009, September 4). Shor's Quantum Factoring Algorithm on a Photonic Chip. *Science* , p. 1221.

Shannon, C. (1949). Communication Theory of Secrecy Systems. *Bell System Technical Journal , 28*, 656-715.

Shor, P. W. (1994). Algorithms for Quantum Computation: Discrete Logarithms and Factoring., *35*, p. 124.

Sugimoto, T., & Yamazaki, K. (2000). A study on secret key reconciliation protocol "cascade". *IEICE Trans. Fundamentals , E83-A* (10), 1987-1991.

Van Assche, G. (2006). *Quantum Cryptography and Secret-Key Distillation.* Cambridge: Cambridge University Press.

Wegman, M. N., & Carter, J. L. (1981). New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences* , 265-279.

Wiesner, S. (1983, Winter-Spring). Conjugate Coding. *ACM SIGACT News - A Special Issue on Cryptography , 15* (1), pp. 78-80.

Yamazaki, K., Nair, R., & Yuen, H. P. (2006). Problems of the CASCADE Protocol and Renyi Entropy Reductions in Classical and Quantum Key Generation. *QCMC* . arXiv:quant-ph/0703012.

Yan, H., Peng, X., Lin, X., Jiang, W., Liu, T., & Guo, H. (2009). Efficiency of Winnow Protocol in Secret Key Reconciliation. *Computer Science and Information Engineering, WRI World Congress*, *7*, pp. 238-242. Los Angeles.

Zhao, F., Fu, M., Wang, F., Lu, Y., Liao, C., & Liu, S. (2007). Error reconciliation for practical quantum cryptography. *Optik , 118* (10), 502-506.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From – To)*
16-06-2011	Master's Thesis	Jan 2011 – May 2011

4. TITLE AND SUBTITLE
Performance Analysis and Optimization of the Winnow Secret Key Reconciliation Protocol

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Kevin C Lustic

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)
AFIT/ENG
2950 Hobson Way
WPAFB, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCO/ENG11-08

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Information withheld

10. SPONSOR/MONITOR'S ACRONYM(S)
Information withheld

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Information withheld

12. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

13. SUPPLEMENTARY NOTES
This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Currently, private communications in public and government sectors rely on methods of cryptographic key distribution that will likely be rendered obsolete the moment a full-scale quantum computer is realized, or efficient classical methods of factoring are discovered. There are alternative methods for distributing secret key material in a "post-quantum" era. One example of a system capable of securely distributing cryptographic key material, known as Quantum Key Distribution (QKD), is secure against quantum factorization techniques as its security rests on generally accepted laws of quantum physics. QKD protocols typically include a phase called "Error Reconciliation", a clear-text classical-channel discussion between legitimate parties of a QKD protocol by which errors introduced in the quantum channel are corrected and the legitimate parties ensure they share identical key material. This work improves one such reconciliation protocol, called *Winnow*, by examining block-size choices for *Winnow* and thus increasing QKD key rate. Block sizes are chosen to maximize the probability that each block contains exactly one error. Further analyses of *Winnow* are provided to characterize the effects of different error distributions on protocol operation and shed light on the time and communication complexities of the *Winnow* secret key reconciliation protocol.

15. SUBJECT TERMS
QKD, quantum key distribution, Winnow, error correction, error reconciliation, adaptive, block size selection

16. SECURITY CLASSIFICATION OF: UNCLASSIFIED			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
REPORT	**ABSTRACT**	**c. THIS PAGE**			Lt Col Jeffrey Humphries
U	**U**	**U**	UU	117	**19b. TELEPHONE NUMBER** *(Include area code)* (937)785-6565x7253 Jeffrey.Humphries@afit.edu

Standard Form 298 (Rev: 8-98)
Prescribed by ANSI Std. Z39-18